IBM VisualAge for C++ for Windows

S33H-5038-00

C Library Reference

Version 3.5

IBM VisualAge for C++ for Windows

S33H-5038-00

C Library Reference

Version 3.5

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

First Edition (February 1996)

This edition applies to Version 3.5 of IBM VisualAge for C++ for Windows (33H4979, 33H4980) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

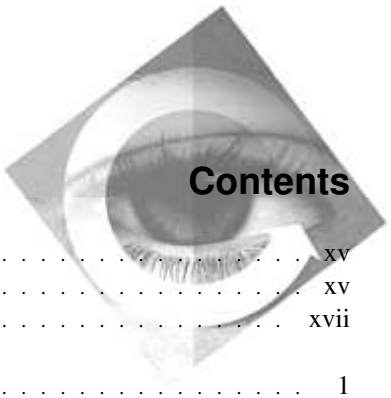
A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1992, 1996. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.



Contents

Notices xv
Programming Interface Information xv
Trademarks and Service Marks xvii

Chapter 1. About This Book 1
Who Should Read This Book 1
Portability Considerations 1
Highlighting Conventions 2
Icons Used in This Book 2
How to Get Help 2
Getting Help Inside VisualAge for C++ 3
Getting Help from the Command Line 4
Getting Help for a Keyword or Construct 4
Online Documents Available in VisualAge for C++ 4

Chapter 2. The C Library 5
Summary of Library Functions 5
Error Handling 5
Process Control 5
File and Directory Management 6
Searching and Sorting 6
Regular Expressions 7
Mathematical 7
Fast RAM Semaphores 9
Floating-Point Unit Control 9
Time Manipulation 9
Date, Time, and Monetary Manipulation 10
Type Conversion 10
Stream Input/Output 11
Low-Level Input/Output 14
Handling Argument Lists 16
Pseudorandom Numbers 16
Dynamic Memory Management 16
Memory Objects 18
Environment Interaction 18
String Operations 19
Character Testing 20
Character Case Mapping 21
Wide Character String Operation Functions 21
Intrinsic Functions 22

© Copyright IBM Corp. 1992, 1996

Functions that Are Inlined when Optimization Is On 22
Functions that Are Always Inlined 23
Differentiating between Memory Management Functions 23
Heap-Specific Functions 24
Debug Functions 24
Heap-Specific Debug Functions 26
Infinity and NaN Support 27
Infinity and NaN in Library Functions 28
Using Low-Level I/O Functions 35

Chapter 3. Library Functions 37
abort — Stop a Program 39
abs — Calculate Integer Absolute Value 40
access — Determine Access Mode 41
acos — Calculate Arccosine 43
_alloca — Temporarily Reserve Storage Block 45
asctime — Convert Time to Character String 47
asin — Calculate Arcsine 49
assert — Verify Condition 51
atan – atan2 — Calculate Arctangent 53
atexit — Record Program Termination Function 54
atof — Convert Character String to Float 56
atoi — Convert Character String to Integer 58
atol — Convert Character String to Long Integer 60
atoll — Convert Character String to Long Long Integer 62
_atold — Convert Character String to Long Double 64
_beginthread — Create New Thread 66
Bessel Functions — Solve Differential Equations 69
bsearch — Search Arrays 71
_cabs — Calculate Absolute Value of Complex Number 73
calloc — Reserve and Initialize Storage 75
ceil — Find Integer >= Argument 77
_cgets — Read String of Characters from Keyboard 78
chdir — Change Current Working Directory 80
_chdrive — Change Current Working Drive 82
chmod — Change File Permission Setting 83
_chsize — Alter Length of File 85
clearerr — Reset Error Indicators 87
_clear87 — Clear Floating-Point Status Word 88
clock — Determine Processor Time 90
close — Close File Associated with Handle 92
_control87 — Set Floating-Point Control Word 94
cos — Calculate Cosine 96

cosh — Calculate Hyperbolic Cosine 97
_cprintf — Print Characters to Screen 98
_cputs — Write String to Screen 99
creat — Create New File 100
_crotl – _crotr — Rotate Bits of Character Value 102
_CRT_init — Initialize DLL Runtime Environment 103
_CRT_term — Terminate DLL Runtime Environment 107
csid — Determine Character Set ID for Multibyte Character 111
_cscanf — Read Data from Keyboard 112
ctime — Convert Time to Character String 114
_cwait — Wait for Child Process 116
__cxchg — Exchange Character Value with Memory 119
_debug_calloc — Allocate and Initialize Memory 124
_debug_free — Release Memory 126
_debug_heapmin — Release Unused Memory in the Default Heap 128
_debug_malloc — Allocate Memory 130
_debug_realloc — Reallocate Memory Block 132
_debug_ucalloc — Reserve and Initialize Memory from User Heap 134
_debug_uheapmin — Release Unused Memory in User Heap 136
_debug_umalloc — Reserve Memory Blocks from User Heap 138
difftime — Compute Time Difference 140
_disable — Disable Interrupts 142
div — Calculate Quotient and Remainder 143
_DLL_InitTerm — Initialize and Terminate DLL Environment 144
_dump_allocated — Get Information about Allocated Memory 154
_dump_allocated_delta — Get Information about Allocated Memory 157
dup — Associate Second Handle with Open File 160
dup2 — Associate Second Handle with Open File 163
_ecvt — Convert Floating-Point to Character 166
_enable — Enable Interrupts 168
_endthread — Terminate Current Thread 169
__eof — Determine End of File 171
erf – erfc — Calculate Error Functions 173
execl - _execvpe — Load and Run Child Process 175
exit — End Program 179
_exit — End Process 180
exp — Calculate Exponential Function 181
fabs — Calculate Floating-Point Absolute Value 182
_facos — Calculate Arccosine 183
_fasin — Calculate Arcsine 185
fclose — Close Stream 187
_fcloseall — Close All Open Streams 188
_fcos — Calculate Cosine 189

_fcossin — Calculate Cosine and Sine 190
_fcvt — Convert Floating-Point to String 192
fdopen — Associates Input Or Output With File 194
feof — Test End-of-File Indicator 197
ferror — Test for Read/Write Errors 198
fflush — Write Buffer to File 199
fgetc — Read a Character 200
_fgetchar — Read Single Character from **stdin** 202
fgetpos — Get File Position 203
fgets — Read a String 205
fgetwc — Read Wide Character from Stream 207
fgetws — Read Wide-Character String from Stream 209
_filelength — Determine File Length 211
fileno — Determine File Handle 213
floor — Integer <= Argument 214
_flushall — Write Buffers to Files 215
fmod — Calculate Floating-Point Remainder 217
fopen — Open Files 218
_fpatan — Calculate Arctangent 221
_fpreset — Reset Floating-Point Unit 222
fprintf — Write Formatted Data to a Stream 224
_fptan — Calculate Tangent 226
fputc — Write Character 227
_fputchar — Write Character 229
fputs — Write String 230
fputwc — Write Wide Character 232
fputws — Write Wide-Character String 234
fread — Read Items 236
free — Release Storage Blocks 238
_freemod — Free User DLL 240
freopen — Redirect Open Files 242
frexp — Separate Floating-Point Value 244
fscanf — Read Formatted Data 245
fseek — Reposition File Position 247
fsetpos — Set File Position 249
_fsin — Calculate Sine 251
_fsincos — Calculate Sine and Cosine 252
_fsqrt — Calculate Square Root 254
fstat — Information about Open File 255
ftell — Get Current Position 257
_ftime — Store Current Time 259
_fullpath — Get Full Path Name of Partial Path 261
fwrite — Write Items 263

_fyl2x — Calculate y * log2(x) 264
_fyl2xp1 — Calculate y * log2(x + 1) 265
_f2xm1 — Calculate (2**x) – 1 266
gamma — Gamma Function 267
_gcvt — Convert Floating-Point to String 268
getc – getchar — Read a Character 270
_getch - _getche — Read Character from Keyboard 272
_getcwd — Get Path Name of Current Directory 274
_getdcwd — Get Full Path Name of Current Directory 276
_getdrive — Get Current Working Drive 278
getenv — Search for Environment Variables 279
getpid — Get Process Identifier 280
gets — Read a Line 281
getsyntx — Return LC_SYNTAX Characters 283
getwc — Read Wide Character from Stream 285
getwchar — Get Wide Character from stdin 287
gmtime — Convert Time 289
_heap_check — Validate Default Memory Heap 291
_heapchk — Validate Default Memory Heap 293
_heapmin — Release Unused Memory from Default Heap 295
_heapset — Validate and Set Default Heap 296
_heap_walk — Return Information about Default Heap 298
hypot — Calculate Hypotenuse 301
iconv — Convert Characters to New Code Set 302
iconv_close — Remove Conversion Descriptor 305
iconv_open — Create Conversion Descriptor 306
_inp — Read Byte from Input Port 308
_inpd — Read Doubleword from Input Port 310
_inpw — Read Unsigned Short from Input Port 312
_interrupt — Call Interrupt Procedure 314
isalnum to isxdigit — Test Integer Value 315
isascii — Test Integer Values 319
isatty — Test Handle for Character Device 321
isblank — Test for Blank Character Classification 323
_iscsym - _iscsymf — Test Integer 325
iswalnum to iswxdigit — Test Wide Integer Value 327
iswblank — Test for Wide Blank Character Classification 331
iswctype — Test for Character Property 333
_itoa — Convert Integer to String 336
_kbhit — Test for Keystroke 337
labs — Calculate Absolute Value of Long Integer 339
ldexp — Multiply by a Power of Two 340
ldiv — Perform Long Division 341

lfind - lsearch — Find Key in Array 342
llabs — Calculate Absolute Value of Long Long Integer 344
lldiv — Perform Long Long Division 345
_llrotl - _llrotr — Rotate Bits of Unsigned Long Long Integer 346
_loadmod — Load DLL 348
localdtconv — Return Date and Time Formatting Convention 350
localeconv — Retrieve Information from the Environment 352
localtime — Convert Time 356
log — Calculate Natural Logarithm 358
log10 — Calculate Base 10 Logarithm 359
longjmp — Restore Stack Environment 360
_lrotl - _lrotr — Rotate Bits of Unsigned Long Value 364
lseek — Move File Pointer 365
_ltoa — Convert Long Integer to String 367
__lxchg — Exchange Integer Value with Memory 369
_makepath — Create Path 374
malloc — Reserve Storage Block 376
_matherr — Process Math Library Errors 378
max — Return Larger of Two Values 381
mblen — Determine Length of Multibyte Character 382
mbrlen — Calculate Length of Multibyte Character 384
mbrtowc — Convert Multibyte Character to Wide Character 386
mbsinit — Test Object for Initial State 388
mbsrtowcs — Convert Multibyte String to Wide-Character String 389
mbstowcs — Convert Multibyte String to Wide-Character String 391
mbtowc — Convert Multibyte Character to Wide Character 393
memccpy — Copy Bytes 395
memchr — Search Buffer 396
memcmp — Compare Buffers 397
memcpy — Copy Bytes 399
memicmp — Compare Bytes 400
memmove — Copy Bytes 402
memset — Set Bytes to Value 403
_mheap — Query Memory Heap for Allocated Object 404
min — Return Lesser of Two Values 406
mkdir — Create New Directory 407
mktime — Convert Local Time 410
modf — Separate Floating-Point Value 412
_msize — Return Number of Bytes Allocated 413
nl_langinfo — Retrieve Locale Information 414
_onexit — Record Termination Function 416
open — Open File 418
_outp — Write Byte to Output Port 421

_outpd — Write Double Word to Output Port 423
_outpw — Write Word to Output Port 425
perror — Print Error Message 427
pow — Compute Power 428
printf — Print Formatted Characters 429
putc – putchar — Write a Character 436
_putch — Write Character to Screen 438
putenv — Modify Environment Variables 439
puts — Write a String 441
putwc — Write Wide Character 442
putwchar — Write Wide Character to stdout 444
qsort — Sort Array 446
raise — Send Signal 448
rand — Generate Random Number 449
read — Read Into Buffer 450
realloc — Change Reserved Storage Block Size 452
regcomp — Compile Regular Expression 455
regerror — Return Error Message for Regular Expression 457
regexec — Execute Compiled Regular Expression 459
regfree — Free Memory for Regular Expression 462
remove — Delete File 463
rename — Rename File 464
rewind — Adjust Current File Position 465
rmdir — Remove Directory 467
_rmem_init — Initialize Memory Functions for Subsystem DLL 470
_rmem_term — Terminate Memory Functions for Subsystem DLL 476
_rmtmp — Remove Temporary Files 482
_rotl - _rotr — Rotate Bits of Unsigned Integer 483
rpmatch — Test for Yes/No Response Match 485
scanf — Read Data 486
_searchenv — Search for File 492
setbuf — Control Buffering 493
_set_crt_msg_handle — Change Runtime Message Output Destination . . . 495
setjmp — Preserve Environment 497
setlocale — Set Locale 499
_setmode — Set File Translation Mode 505
setvbuf — Control Buffering 508
signal — Handle Interrupt Signals 510
sin — Calculate Sine 514
sinh — Calculate Hyperbolic Sine 515
_sopen — Open Shared File 516
_spawnl - _spawnvpe — Start and Run Child Processes 519
_splitpath — Decompose Path Name 523

sprintf — Print Formatted Data to Buffer 525
sqrt — Calculate Square Root 527
srand — Set Seed for rand Function 528
_srotl - _srotr — Rotate Bits of Unsigned Short Value 529
sscanf — Read Data 530
stat — Get Information about File or Directory 532
_status87 — Get Floating-Point Status Word 534
strcat — Concatenate Strings 536
strchr — Search for Character 537
strcmp — Compare Strings 538
strcmpi — Compare Strings Without Case Sensitivity 540
strcoll — Compare Strings Using Collation Rules 542
strcpy — Copy Strings 544
strcspn — Compare Strings for Substrings 546
_strdate — Copy Current Date 548
strdup — Duplicate String 549
strerror — Set Pointer to Runtime Error Message 550
_strerror — Set Pointer to System Error String 551
strfmon — Convert Monetary Value to String 553
strftime — Convert to Formatted Time 557
stricmp — Compare Strings as Lowercase 562
strlen — Determine String Length 564
strlwr — Convert Uppercase to Lowercase 565
strncat — Concatenate Strings 566
strncmp — Compare Strings 568
strncpy — Copy Strings 570
strnicmp — Compare Strings Without Case Sensitivity 572
strnset – strset — Set Characters in String 574
strpbrk — Find Characters in String 575
strptime — Convert to Formatted Date and Time 576
strrchr — Find Last Occurrence of Character in String 581
strrev — Reverse String 583
strspn — Search Strings 585
strstr — Locate Substring 587
_strtime — Copy Time 589
strtod — Convert Character String to Double 590
strtok — Tokenize String 592
strtol — Convert Character String to Long Integer 594
strtold — Convert String to Long Double 596
strtoll — Convert Character String to Long Long Integer 598
strtoul — Convert String Segment to Unsigned Integer 600
strtoull — Convert String Segment to Unsigned Long Long Integer 602
strupr — Convert Lowercase to Uppercase 604

strxfrm — Transform String 605
swab — Swap Adjacent Bytes 608
swprintf — Format and Write Wide Characters to Buffer 609
swscanf — Read Wide Characters from Buffer 611
system — Invoke the Command Processor 613
__sxchg — Exchange Integer Value with Memory 615
tan — Calculate Tangent 617
tanh — Calculate Hyperbolic Tangent 618
_tell — Get Pointer Position 619
tempnam — Produce Temporary File Name 621
_threadstore — Access Thread-Specific Storage 623
time — Determine Current Time 625
tmpfile — Create Temporary File 626
tmpnam — Produce Temporary File Name 627
_toascii - _tolower - _toupper — Convert Character 628
tolower - toupper — Convert Character Case 631
towlower – towupper — Convert Wide Character Case 632
tzset — Assign Values to Locale Information 634
_uaddmem — Add Memory to a Heap 637
_ucalloc — Reserve and Initialize Memory from User Heap 641
_uclose — Close Heap from Use 643
_ucreate — Create a Memory Heap 646
_udefault — Change the Default Heap 650
_udestroy — Destroy a Heap 653
_udump_allocated — Get Information about Allocated Memory in Heap . . . 656
_udump_allocated_delta — Get Information about Allocated Memory in Heap 659
_uheap_check — Validate User Memory Heap 662
_uheapchk — Validate Memory Heap 665
_uheapmin — Release Unused Memory in User Heap 667
_uheapset — Validate and Set Memory Heap 669
_uheap_walk — Return Information about Memory Heap 671
_ultoa — Convert Unsigned Long Integer to String 675
_ulltoa — Convert Unsigned Long Long Integer to String 676
_umalloc — Reserve Memory Blocks from User Heap 677
umask — Sets File Mask of Current Process 679
ungetc — Push Character onto Input Stream 681
_ungetch — Push Character Back to Keyboard 683
ungetwc — Push Wide Character onto Input Stream 685
unlink — Delete File 688
_uopen — Open Heap for Use 689
_ustats — Get Information about Heap 691
utime — Set Modification Time 693
va_arg - va_end - va_start — Access Function Arguments 696

vfprintf — Print Argument Data to Stream 698
vprintf — Print Argument Data 700
vsprintf — Print Argument Data to Buffer 702
vswprintf — Format and Write Wide Characters to Buffer 704
wcrtomb — Convert Wide Character to Multibyte Character 706
wcscat — Concatenate Wide-Character Strings 708
wcschr — Search for Wide Character 709
wcscmp — Compare Wide-Character Strings 711
wcscoll — Compare Wide-Character Strings 713
wcscpy — Copy Wide-Character Strings 715
wcscspn — Find Offset of First Wide-Character Match 717
wcsftime — Convert to Formatted Date and Time 719
wcsid — Determine Character Set ID for Wide Character 721
wcslen — Calculate Length of Wide-Character String 722
wcsncat — Concatenate Wide-Character Strings 723
wcsncmp — Compare Wide-Character Strings 725
wcsncpy — Copy Wide-Character Strings 727
wcspbrk — Locate Wide Characters in String 729
wcsrchr — Locate Wide Character in String 731
wcsrtombs — Convert Wide-Character String to Multibyte String 733
wcsspn — Search Wide-Character Strings 735
wcsstr — Locate Wide-Character Substring 736
wcstod — Convert Wide-Character String to Double 737
wcstok — Tokenize Wide-Character String 739
wcstol — Convert Wide-Character to Long Integer 741
wcstombs — Convert Wide-Character String to Multibyte String 743
wcstoul — Convert Wide-Character String to Unsigned Long 745
wcswcs — Locate Wide-Character Substring 747
wcswidth — Determine Display Width of Wide-Character String 748
wcsxfrm — Transform Wide-Character String 749
wctob — Convert Wide Character to Byte 752
wctomb — Convert Wide Character to Multibyte Character 753
wctype — Get Handle for Character Property Classification 755
wcwidth — Determine Display Width of Wide Character 758
write — Writes from Buffer to File 759

Chapter 4. Include Files 761
<assert.h> 761
<builtin.h> 761
<conio.h> 762
<ctype.h> 762
<direct.h> 762
<errno.h> 763

<fcntl.h> . . . 763
<float.h> . . . 763
<iconv.h> . . . 763
<io.h> . . . 764
<langinfo.h> . . . 764
<lc_core.h> . . . 765
<limits.h> . . . 766
<localdef.h> . . . 766
<locale.h> . . . 766
<malloc.h> . . . 769
<math.h> . . . 770
<memory.h> . . . 771
<monetary.h> . . . 771
<nl_types.h> . . . 771
<process.h> . . . 772
<regex.h> . . . 772
<search.h> . . . 772
<setjmp.h> . . . 773
<share.h> . . . 773
<signal.h> . . . 773
<stdarg.h> . . . 773
<stddef.h> . . . 773
<stdio.h> . . . 774
<stdlib.h> . . . 775
<string.h> . . . 777
<sys\stat.h> . . . 778
<sys\timeb.h> . . . 778
<sys\types.h> . . . 778
<sys\utime.h> . . . 778
<time.h> . . . 779
<umalloc.h> . . . 779
<variant.h> . . . 780
<wchar.h> . . . 780
<wcstr.h> . . . 782
<wctype.h> . . . 782

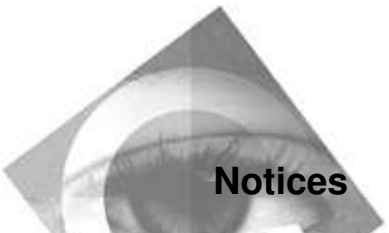
Appendix A. VisualAge for C++ Functions and Macros . . . 783
Functions and Macros . . . 783
Predefined Macros . . . 808

Glossary . . . 811

Bibliography . . . 823

The IBM VisualAge for C++ Library 823
C and C++ Related Publications 823
Non-IBM Publications 823

Index 825



Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

This book is intended to help you create programs using VisualAge for C++ product. It primarily documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge for C++ product.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge for C++ compiler and debugger.

However, this book also documents Diagnosis, Modification, and Tuning Information. Diagnosis, Modification, and Tuning Information is provided to help you debug your programs.

Warning: Do not use this Diagnosis, Modification, and Tuning Information as a programming interface because it is subject to change.

© Copyright IBM Corp. 1992, 1996

Diagnosis, Modification, and Tuning Information is identified where it occurs by an introductory statement to a chapter or section.

Trademarks and Service Marks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

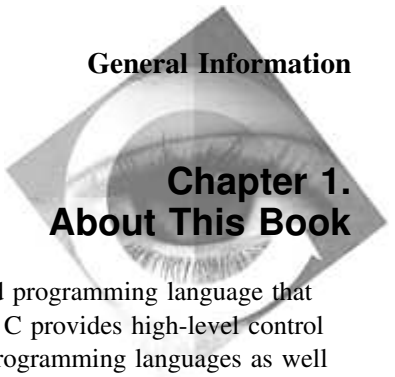
AIX	SAA
IBM	Systems Application Architecture
IBMLink	VisualAge
PROFS	WorkFrame

Windows is a trademark of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

IBM's VisualAge products and services are not associated with or sponsored by Visual Edge Software, Ltd..

General Information



Chapter 1. About This Book

The C language is a general-purpose, function-oriented programming language that you can use to create applications quickly and easily. C provides high-level control statements and data types similar to other structured programming languages as well as many of the benefits of a low-level language. Portable code is easily written in the C language.

This book describes the library functions provided by the VisualAge for C++ product, including those defined by:

- The American National Standards Institute C Standard and International Standards Organization, ANSI/ISO 9899-1990[1992], and the amendment ISO/IEC 9899:1990/Amendment 1:1993(E)
- The ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard
- The X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.

Unless explicitly indicated otherwise, all of the library functions described in this book are also available to C++ programs.

Who Should Read This Book

This book is written for application programmers who want to use the functions and macros provided by VisualAge for C++ to develop and run C and C++ applications on the Windows platform. It assumes you have a working knowledge of the C programming language and the Windows operating system.

Portability Considerations

If you will be using VisualAge for C++ to develop applications that will also be compiled and run on other systems, you should refer to the current language standards described in Chapter 1, “About This Book.”

The language level is given for each function in Chapter 3, “Library Functions” on page 37. When creating programs to conform strictly to language standards such as ANSI/ISO or POSIX, do **not** use the functions described as extensions in this book.

© Copyright IBM Corp. 1992, 1996

Highlighting Conventions

The following highlighting conventions are used in this book:

Bold	Identifies commands and language keywords.
Italics	Identify parameters whose actual names or values are to be supplied by the programmer.
Example	Identifies function names, examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Icons Used in This Book

The VisualAge for C++ library uses icons to let you quickly scan pages for key concepts, examples, cross-references, and other information.

This icon identifies examples that illustrate how to use a particular language feature or other concept presented in the book.

This icon identifies cross-references to related information in this or other books. The icon may appear in the left margin where a number of cross-references are collected, or in miniature form within the text of a paragraph (like this:) where only one or two cross-references are shown.

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++:

Online documents

These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++. For your convenience, the online documents are presented in:

Standard format (.INF files). See "Getting Help Inside VisualAge for C++" on page 3 for instructions on opening standard format documents from inside VisualAge for C++. See "Getting Help from the Command Line" on page 4 for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ documents that are available in standard format, see "Online Documents Available in VisualAge for C++" on page 4.

Contextual help

Contextual help is available throughout VisualAge for C++. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I **help**

Many of the common tasks that you want to perform with VisualAge for C++ are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++

All three kinds of help are available directly within the VisualAge for C++ interface:

To get general contextual help for the component of VisualAge for C++ that you are using, press **F1** anywhere in the window.

To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.

To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:

- **Help Index**, an alphabetical list of all of the help topics that are available from this window
- **General Help**, overall help for the window
- **Using Help**, general information about the help facility
- **How Do I...**, the How Do I help for the component
- **Product Information**, a dialog that shows the level of VisualAge for C++ being used

In addition, there are selections that let you open all of online documents that are available in VisualAge for C++.

To get detailed information, open the **Online Information** notebook in the VisualAge for C++ folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the iview command. The installation routine stores the online document files in the \IBMCPPW\HELP directory. To view the *Language Reference*, for example, make C:\IBMCPPW\HELP your current directory (substituting the drive where you installed VisualAge for C++ for C:) and enter the following command:

```
IVIEW  CPPLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the *Language Reference*, you can enter the following command:

```
IVIEW CPPLNG.INF OPERATOR PRECEDENCE
```

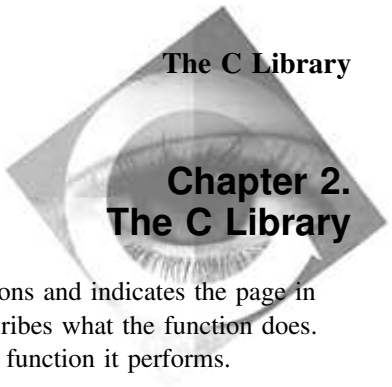
Getting Help for a Keyword or Construct

If you are editing a file using the Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing **Ctrl-H**. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing **Ctrl-H**.

Online Documents Available in VisualAge for C++

The following documents are available in standard format:

Building VisualAge for C++ Parts for Fun and Profit	*Open Class Library Reference*
C Library Reference	*Open Class Library User's Guide*
Editor Command Reference	*Programming Guide*
Frequently Asked Questions	*SOM Programming Guide*
Installation Guide and Product Overview	*SOM Programming Reference*
IPF User's Guide	*User's Guide*
IPF Programmer's Guide and Reference	*Visual Builder User's Guide*
Language Reference	*Visual Builder Parts Reference*



Chapter 2. The C Library

This chapter summarizes the available C library functions and indicates the page in this book where each is described. It also briefly describes what the function does. Each library function is listed according to the type of function it performs.

Summary of Library Functions

Error Handling

Function	Header File	Page	Description
assert	assert.h	51	Prints diagnostic messages.
atexit	stdlib.h	54	Registers a function to be executed at program termination.
clearerr	stdio.h	87	Resets error indicators.
ferror	stdio.h	198	Tests the error indicator for a specified stream.
_matherr	math.h	378	Processes errors generated by the functions in the math library.
perror	stdio.h	427	Prints an error message to **stderr**.
raise	signal.h	448	Initiates a signal.
_set_crt_msg_handle	stdio.h	495	Changes the file handle to which runtime messages are sent.
signal	signal.h	510	Allows handling of an interrupt signal from the operating system.
strerror	string.h	550	Sets pointer to system error message.
_strerror	string.h	551	Tests for system error.

Process Control

Function	Header File	Page	Description
_beginthread	stdlib.h process.h	66	Creates a new thread.
_cwait	process.h	116	Delays the completion of a parent process until a child process ends.
_disable	builtin.h	142	Disables interrupts.
_enable	builtin.h	168	Enables interrupts.
_endthread	stdlib.h process.h	169	Terminates a thread.
execl – _execvpe	process.h	175	Load and run child processes.
_exit	stdlib.h process.h	180	Ends the calling process without calling other functions.
_freemod	stdlib.h	240	Frees all references to a DLL for the calling process.
getpid	process.h	280	Gets the process identifier that identifies the calling process.
_interrupt	builtin.h	314	Calls an interrupt procedure.
_loadmod	stdlib.h	348	Loads a user-created DLL for the calling process.

© Copyright IBM Corp. 1992, 1996

Function	Header File	Page	Description
_onexit	stdlib.h	416	Records the address of a function to call when the program ends.
putenv	stdlib.h	439	Adds new environment variables or modifies the values of those already existing.
_searchenv	stdlib.h	492	Searches a specified environment for a target file.
_spawnl – _spawnvpe	process.h	519	Start and run child processes.
_threadstore	stdlib.h	623	Accesses a thread-specific storage space.

File and Directory Management

Function	Header File	Page	Description
chdir	direct.h	80	Changes the current working directory to a specified directory.
_chdrive	direct.h	82	Changes the current working drive to a specified drive.
fstat	sys\stat.h	255	Gets and stores information about the open file.
_fullpath	stdlib.h	261	Gets and stores the full path name of a given partial path.
_getcwd	direct.h	274	Gets the full path name of the current working directory.
_getdcwd	direct.h	276	Gets the full path name for the current directory of a specified drive.
_getdrive	direct.h	278	Returns an integer corresponding to the letter representing the current drive.
_makepath	stdlib.h	374	Creates a single path name.
mkdir	direct.h	407	Creates a new directory.
rmdir	direct.h	467	Deletes a directory.
_splitpath	stdlib.h	523	Decomposes a path name into its four components.
stat	sys\stat.h	532	Stores information about a file or directory in a structure.

Searching and Sorting

Function	Header File	Page	Description
bsearch	stdlib.h search.h	71	Performs a binary search of a sorted array.
lfind	search.h	342	Performs a linear search for a value in an array.
lsearch	search.h	342	Performs a linear search for a value in an array.
qsort	stdlib.h search.h	446	Performs a quick sort on an array of elements.

Regular Expressions

Function	Header File	Page	Description
regcomp	regex.h	455	Compiles a source regular expression into an executable version.
regerror	regex.h	457	Finds the description for the error code for a regular expression.
regexec	regex.h	459	Compares a string with a regular expression.
regfree	regex.h	462	Frees the memory for a regular expression, removing the compiled regular expression.

Mathematical

Function	Header File	Page	Description
abs	stdlib.h	40	Calculates the absolute value of an integer.
_cabs	math.h	73	Calculates the absolute value of a complex number.
ceil	math.h	77	Calculates the double value representing the smallest integer that is greater than or equal to a number.
div	stdlib.h	143	Calculates the quotient and remainder of an integer.
erf	math.h	173	Calculates the error function.
erfc	math.h	173	Calculates the error function for large numbers.
exp	math.h	181	Calculates an exponential function.
fabs	math.h	182	Calculates the absolute value of a floating-point number.
floor	math.h	214	Calculates the double value representing the largest integer that is less than or equal to a number.
fmod	math.h	217	Calculates the floating point remainder of one argument divided by another.
frexp	math.h	244	Separates a floating-point number into its mantissa and exponent.
_fsqrt	math.h	254	Calculates the square root of a number.
_fyl2x	builtin.h	264	Calculates the base-2 logarithm of *x* and multiplies it by *y*.
_fyl2xp1	builtin.h	265	Calculates the base-2 logarithm of *x*+1 and multiplies it by *y*
_f2xm1	builtin.h	266	Calculates ((2**x)-1).
gamma	math.h	267	Calculates the gamma function.
hypot	math.h	301	Calculates the hypotenuse.
labs	stdlib.h	339	Calculates the absolute value of a long integer.
ldexp	math.h	340	Multiplies a floating-point number by an integral power of 2.
ldiv	stdlib.h	341	Calculates the quotient and remainder of a long integer.
llabs	stdlib.h	344	Calculates the absolute value of a long long integer.
lldiv	stdlib.h	345	Calculates the quotient and remainder of a long long integer.
log	math.h	358	Calculates natural logarithm.
log10	math.h	359	Calculates base 10 logarithm.
max	stdlib.h	381	Compares two values and returns the larger of the two.
min	stdlib.h	406	Compares two values and returns the smaller of the two.
modf	math.h	412	Calculates the signed fractional portion of the argument.
pow	math.h	428	Calculates the value of an argument raised to a power.

Function	Header File	Page	Description
sqrt	math.h	527	Calculates the square root of a number.

Trigonometric Functions

Function	Header File	Page	Description
acos	math.h	43	Calculates the arccosine.
asin	math.h	49	Calculates the arc sine.
atan	math.h	53	Calculates the arctangent.
atan2	math.h	53	Calculates the arctangent.
cos	math.h	96	Calculates the cosine.
cosh	math.h	97	Calculates the hyperbolic cosine.
_facos	builtin.h math.h	183	Calculates the arccosine.
_fasin	builtin.h math.h	185	Calculates the arcsine.
_fcos	builtin.h math.h	189	Calculates the cosine.
_fcossin	builtin.h	190	Calculates the cosine and stores the sine.
_fpatan	builtin.h math.h	221	Calculates the arctangent.
_fptan	builtin.h math.h	226	Calculates the tangent.
_fsin	builtin.h math.h	251	Calculates the sine.
_fsincos	builtin.h	252	Calculates the sine and stores the cosine.
sin	math.h	514	Calculates the sine.
sinh	math.h	515	Calculates the hyperbolic sine.
tan	math.h	617	Calculates the tangent.
tanh	math.h	618	Calculates the hyperbolic tangent.

Bit Rotation

Function	Header File	Page	Description
_crotl	stdlib.h builtin.h	102	Rotates a character value to the left by a specified number of bits.
_crotr	stdlib.h builtin.h	102	Rotates a character value to the right by a specified number of bits.
_llrotl	stdlib.h builtin.h	346	Rotates a long long integer value to the left by a specified number of bits.
_llrotr	stdlib.h builtin.h	346	Rotates a long long integer value to the right by a specified number of bits.
_lrotl	stdlib.h builtin.h	364	Rotates a long integer value to the left by a specified number of bits.
_lrotr	stdlib.h builtin.h	364	Rotates a long integer value to the right by a specified number of bits.

Function	Header File	Page	Description
_rotl	stdlib.h builtin.h	483	Rotates an integer value to the left by a specified number of bits.
_rotr	stdlib.h builtin.h	483	Rotates an integer value to the right by a specified number of bits.
_srotl	stdlib.h builtin.h	529	Rotates a short integer value to the left by a specified number of bits.
_srotr	stdlib.h builtin.h	529	Rotates a short integer value to the right by a specified number of bits.

Bessel Functions

Function	Header File	Page	Description
_j0	math.h	69	0 order differential equation of the first kind.
_j1	math.h	69	1st order differential equation of the first kind.
_jn	math.h	69	*n*th order differential equation of the first kind.
_y0	math.h	69	0 order differential equation of the second kind.
_y1	math.h	69	1st order differential equation of the second kind.
_yn	math.h	69	*n*th order differential equation of the second kind.

Fast RAM Semaphores

Function	Header File	Page	Description
__cxchg	builtin.h	119	Exchanges a character value with a previously stored value.
__lxchg	builtin.h	369	Exchanges a long integer value with a previously stored value.
__sxchg	builtin.h	615	Exchanges a short integer value with a previously stored value.

Floating-Point Unit Control

Function	Header File	Page	Description
_clear87	float.h builtin.h	88	Gets the floating-point status word and clears it.
_control87	float.h builtin.h	94	Gets the current floating-point control word and sets it.
_fpreset	float.h	222	Resets the floating-point unit to the default state.
_status87	float.h builtin.h	534	Gets the current floating-point status word.

Time Manipulation

Date, Time, and Monetary Manipulation

Function	Header File	Page	Description
asctime	time.h	47	Converts time stored as a structure to a character string in storage.
clock	time.h	90	Determines processor time.
ctime	time.h	114	Converts time stored as a long value to a character string.
difftime	time.h	140	Calculates the difference between two times.
_ftime	sys\timeb.h	259	Obtains the current time and stores it in a structure.
gmtime	time.h	289	Converts time to Coordinated Universal Time.
localtime	time.h	356	Converts time to local time.
mktime	time.h	410	Converts local time into calendar time.
_strdate	time.h	548	Stores the current date in a buffer.
strfmon	monetary.h	553	Converts a monetary value to a formatted string.
strftime	time.h	557	Converts time to a multibyte character string.
strptime	time.h	576	Converts time stored as a character string to a structure.
_strtime	time.h	589	Copies the current time into a buffer.
time	time.h	625	Returns the time in seconds.
tzset	time.h	634	Changes the time zone and daylight saving time zone values.
utime	sys\utime.h	693	Sets the modification time for a file.
wcsftime	wchar.h	719	Converts the time and date to a wide character string.

Type Conversion

Function	Header File	Page	Description
atof	stdlib.h	56	Converts a character string to a floating-point value.
atoi	stdlib.h	58	Converts a character string to an integer.
atol	stdlib.h	60	Converts a character string to a long integer.
_atold	stdlib.h math.h	64	Converts a character string to a long double value.
atoll	stdlib.h	62	Converts a character string to a long long integer.
_ecvt	stdlib.h	166	Converts a floating-point number to a character string.
_fcvt	stdlib.h	192	Converts a floating-point number to a character string, rounding according to the FORTRAN F format.
_gcvt	stdlib.h	268	Converts a floating-point value to a character string, rounding according to the FORTRAN F or FORTRAN E formats.
_itoa	stdlib.h	336	Converts the digits of an integer to a character string.
_ltoa	stdlib.h	367	Converts the digits of a long integer to a character string.
strtod	stdlib.h	590	Converts a character string to a double value.
strtol	stdlib.h	594	Converts a character string to a long integer.
strtold	stdlib.h	596	Converts a character string to a long double value.
strtoll	stdlib.h	594	Converts a character string to a long long integer.
strtoul	stdlib.h	600	Converts a string to an unsigned long integer.
strtoull	stdlib.h	600	Converts a string to an unsigned long long integer.
_ultoa	stdlib.h	675	Converts the values of an unasigned long value to a character string.

Function	Header File	Page	Description
_ulltoa	stdlib.h	675	Converts the values of an unasigned long long value to a character string.
wcstod	wchar.h	737	Converts a wide character string to a double value.
wcstol	wchar.h	741	Converts a wide character string to a long integer.
wcstoul	wchar.h	745	Converts a wide character string to an unsigned long integer.
wctob	wchar.h	752	Converts a wide character to a single-byte character.

Multibyte and Wide-Character Type Conversion

Function	Header File	Page	Description
mbrtowc	wchar.h	386	Converts a multibyte character to a wide character (**wchar_t**); restartable version of mbtowc.
mbsrtowcs	wchar.h	389	Converts a multibyte character string to a wide character (**wchar_t**) string; restartable version of mbstowcs.
mbstowcs	stdlib.h	391	Converts a multibyte character string to a wide character (**wchar_t**) string.
mbtowc	stdlib.h	393	Converts a multibyte character to a wide character (**wchar_t**).
wcrtomb	wchar.h	706	Converts a wide character (**wchar_t**) to a multibyte character; restartable version of wctomb.
wcsrtombs	wchar.h	733	Converts a wide character (**wchar_t**) string to a multibyte character string; restartable version of wctomb.
wcstombs	stdlib.h	743	Converts a wide character (**wchar_t**) string to a multibyte character string.
wctomb	stdlib.h	753	Converts a wide character (**wchar_t**) to a multibyte character.

Stream Input/Output

Formatted Input/Output

Function	Header File	Page	Description
fprintf	stdio.h	224	Formats and prints characters to the output stream.
fscanf	stdio.h	245	Reads data from a stream into locations given by arguments.
printf	stdio.h	429	Formats and prints characters to **stdout**.
scanf	stdio.h	486	Reads data from **stdin** into locations given by arguments.
sprintf	stdio.h	525	Formats and writes characters to a buffer.
sscanf	stdio.h	530	Reads data from a buffer into locations given by arguments.
swprintf	wchar.h	609	Formats and writes wide characters to a buffer.
swscanf	wchar.h	611	Reads wide characters from a buffer into locations given by arguments.
vfprintf	stdarg.h stdio.h	698	Formats and prints characters to the output stream using a variable number of arguments.
vprintf	stdarg.h stdio.h	700	Formats and writes characters to **stdout** using a variable number of arguments.

Function	Header File	Page	Description
vsprintf	stdarg.h stdio.h	702	Formats and writes characters to a buffer using a variable number of arguments.
vswprintf	wchar.h stdarg.h	704	Formats and writes wide characters to a buffer using a variable number of arguments.

Character and String Input/Output

Function	Header File	Page	Description
fgetc	stdio.h	200	Reads a character from a specified input stream.
_fgetchar	stdio.h	202	Reads a character from the standard input stream.
fgets	stdio.h	205	Reads a string from a specified input stream.
fputc	stdio.h	227	Prints a character to a specified output stream.
_fputchar	stdio.h	229	Prints a character to the standard output stream.
fputs	stdio.h	230	Prints a string to a specified output stream.
getc	stdio.h	270	Reads a character from a specified input stream.
getchar	stdio.h	270	Reads a character from **stdin**.
gets	stdio.h	281	Reads a line from **stdin**.
putc	stdio.h	436	Prints a character to a specified output stream.
putchar	stdio.h	436	Prints a character to **stdout**.
puts	stdio.h	441	Prints a string to **stdout**.
ungetc	stdio.h	681	Pushes a character back onto a specified input stream.

Wide Character and String Input/Output

Function	Header File	Page	Description
fgetwc	stdio.h wchar.h	207	Reads a wide character from a specified input stream.
fgetws	stdio.h wchar.h	207	Reads a wide string from a specified input stream.
fputwc	stdio.h wchar.h	232	Writes a wide character to a specified output stream.
fputws	stdio.h wchar.h	234	Writes a wide character string to a specified output stream.
getwc	stdio.h wchar.h	285	Reads a wide character from a specified input stream.
getwchar	wchar.h	287	Reads a wide character from **stdin**.
putwc	stdio.h wchar.h	442	Writes a wide character to a specified output stream.
putwchar	wchar.h	444	Writes a wide character to **stdout**.
ungetwc	stdio.h wchar.h	685	Pushes a wide character back onto a specified input stream.

Direct Input/Output

Function	Header File	Page	Description
clearerr	stdio.h	87	Resets error indicators.
feof	stdio.h	197	Tests end-of-file indicator for stream input.
ferror	stdio.h	198	Tests the error indicator for a specified stream.
fread	stdio.h	236	Reads items from a specified input stream.
fwrite	stdio.h	263	Writes items to a specified output stream.

File Positioning

Function	Header File	Page	Description
fgetpos	stdio.h	203	Gets the current position of the file pointer.
fseek	stdio.h	247	Moves the file pointer to a new location.
fsetpos	stdio.h	249	Moves the file pointer to a new location.
ftell	stdio.h	257	Gets the current position of the file pointer.
lseek	io.h	365	Moves a file pointer to a new location.
rewind	stdio.h	465	Repositions the file pointer to the beginning of the file.

File Access

Function	Header File	Page	Description
fclose	stdio.h ifs.h	187	Closes a specified stream.
_fcloseall	stdio.h	188	Closes all open streams, except the standard streams.
fdopen	stdio.h	194	Associates an input or output stream with a file.
fflush	stdio.h	199	Causes the system to write the contents of a buffer to a file.
_flushall	stdio.h	215	Writes the contents of buffers to files.
fopen	stdio.h	218	Opens a specified stream.
freopen	stdio.h	242	Closes a file and reassigns a stream.
setbuf	stdio.h	493	Allows control of buffering.
_setmode	io.h	505	Sets the translation mode of a file.
setvbuf	stdio.h	508	Controls buffering and buffer size for a specified stream.

File Operations

Function	Header File	Page	Description
fileno	stdio.h	213	Determines the file handle.
remove	stdio.h	463	Deletes a specified file.
rename	stdio.h	464	Renames a specified file.
_rmtmp	stdio.h	482	Closes and deletes temporary files.
tempnam	stdio.h	621	Creates a temporary file name in another directory.
tmpfile	stdio.h	626	Creates a temporary file and returns a pointer to that file.
tmpnam	stdio.h	627	Produces a temporary file name.
umask	io.h	679	Sets the file permission mask of the executing process environment.
unlink	stdio.h	688	Deletes a file.

Low-Level Input/Output

Port Input/Output

Function	Header File	Page	Description
_inp	conio.h builtin.h	308	Reads a byte value from a specified input port.
_inpd	conio.h builtin.h	310	Reads a 4-byte value from a specified input port.
_inpw	conio.h builtin.h	312	Reads a 2-byte value from a specified input port.
_outp	conio.h builtin.h	421	Writes a byte value to a specified output port.
_outpd	conio.h builtin.h	423	Writes a 4-byte value to a specified output port.
_outpw	conio.h builtin.h	425	Writes a 2-byte value to a specified output port.
umask	io.h	679	Sets the file permission mask of the executing process environment.

Character and String Input/Output

Function	Header File	Page	Description
_cgets	conio.h	78	Reads a string from the keyboard into locations given by arguments.
_cprintf	conio.h	98	Formats and sends a series of characters and values to the screen.
_cputs	conio.h	99	Writes a string directly to the screen.
_cscanf	conio.h	112	Reads data from the keyboard into locations given by arguments.
_getch	conio.h	272	Reads a single character from the keyboard.
_getche	conio.h	272	Reads a single character from the keyboard and displays it.
_kbhit	conio.h	337	Tests if a key has been pressed on the keyboard.

Function	Header File	Page	Description
_putch	conio.h	438	Writes a character to the screen.
_ungetch	conio.h	683	Pushes a character back to the keyboard.

Direct Input/Output

Function	Header File	Page	Description
read	io.h	450	Reads bytes from a file into a buffer.
write	io.h	759	Writes bytes from a buffer into a file.

File Positioning

Function	Header File	Page	Description
__eof	io.h	171	Determines whether the file pointer has reached the end of the file.
_tell	io.h	619	Gets the current position of a file pointer.

File Access

Function	Header File	Page	Description
access	io.h	41	Determines whether the given file exists and whether you can gain access to it.
chmod	io.h	83	Changes the permission setting of a file.
close	io.h	92	Closes a file associated with the handle.
creat	io.h	100	Creates a new file or opens and truncates an existing file.
dup	io.h	160	Associates a second file handle with an open file.
dup2	io.h	163	Associates a second file handle, with possibly different attributes, with an open file.
isatty	io.h	321	Determines whether the handle is associated with a character device.
open	io.h	418	Opens a file and prepares it for subsequent reading and writing.
_sopen	io.h	516	Opens a file and prepares it for subsequent shared reading or writing.

File Operations

Function	Header File	Page	Description
_chsize	io.h	85	Lengthens or cuts off the file to a specified length.
_filelength	io.h	211	Returns the length of a file.

Handling Argument Lists

Function	Header File	Page	Description
va_arg	stdarg.h	696	Allows access to variable number of function arguments.
va_end	stdarg.h	696	Allows access to variable number of function arguments.
va_start	stdarg.h	696	Allows access to variable number of function arguments.
vfprintf	stdarg.h stdio.h	698	Formats and prints characters to the output stream using a variable number of arguments.
vprintf	stdarg.h stdio.h	700	Formats and writes characters to **stdout** using a variable number of arguments.
vsprintf	stdarg.h stdio.h	702	Formats and writes characters to a buffer using a variable number of arguments.
vswprintf	wchar.h stdarg.h	704	Formats and writes wide characters to a buffer using a variable number of arguments.

Pseudorandom Numbers

Function	Header File	Page	Description
rand	stdlib.h	449	Returns a pseudorandom integer.
srand	stdlib.h	528	Sets the starting point for pseudorandom numbers.

Dynamic Memory Management

Allocating and Freeing Memory

Function	Header File	Page	Description
_alloca	stdlib.h malloc.h builtin.h	45	Temporarily allocates storage space from the program's stack.
calloc	stdlib.h malloc.h	75	Reserves storage space for an array and initializes the values of all elements to 0.
_debug_calloc	stdlib.h malloc.h	124	Allocates memory and initializes it to 0, checks the heap, and stores information about the allocation.
_debug_free	stdlib.h malloc.h	126	Releases memory, checks the heap, and stores information about the operation.
_debug_heapmin	stdlib.h malloc.h	128	Returns unused memory in the heap to the operating system, and stores information about the operation.
_debug_malloc	stdlib.h malloc.h	130	Allocates memory and initializes it to 0xAA, checks the heap, and stores information about the allocation.
_debug_realloc	stdlib.h malloc.h	132	Changes the size of an allocated memory block, sets unused memory to 0xAA, checks the heap, and stores information about the operation.
_debug_ucalloc	stdlib.h malloc.h	134	Allocates memory from a specified heap and initializes it to 0, checks the heap, and stores information about the allocation.

Function	Header File	Page	Description
_debug_uheapmin	stdlib.h malloc.h	136	Returns unused memory from a specified heap to the operating system, and stores information about the operation.
_debug_umalloc	stdlib.h malloc.h	138	Allocates memory from a specified heap and initializes it to 0xAA, checks the heap, and stores information about the allocation.
free	stdlib.h malloc.h	238	Releases memory blocks.
_heapmin	stdlib.h malloc.h	295	Returns all unused memory from the runtime heap to the operating system.
malloc	stdlib.h malloc.h	376	Allocates memory blocks.
realloc	stdlib.h malloc.h	452	Changes storage size allocated for an object.

Heap Information and Checking

Function	Header File	Page	Description
_dump_allocated	stdlib.h malloc.h	154	Writes information about currently allocated memory blocks.
_dump_allocated_delta	stdlib.h malloc.h	157	Writes information about currently allocated memory blocks since the last call for this information.
_heap_check	stdlib.h malloc.h	291	Validates all allocated memory blocks.
_heapchk	malloc.h	293	Validates all allocated and freed objects on the default heap.
_heapset	malloc.h	296	Validates all allocated and freed objects on the default heap, and sets all free memory to a specified value.
_heap_walk	malloc.h	298	Returns information about allocated and freed objects on the default heap.
_mheap	umalloc.h	404	Finds out which heap an object was allocated from.
_msize	stdlib.h malloc.h	413	Returns the size of an allocated block.
_udump_allocated	umalloc.h	656	Writes information about currently allocated memory blocks from a specified heap.
_udump_allocated_delta	umalloc.h	659	Writes information about currently allocated memory blocks from a specified heap since the last call for this information.
_uheap_check	umalloc.h	662	Validates all allocated memory blocks from a specified heap.
_uheapchk	umalloc.h	665	Validates all allocated and freed objects on a specified heap.
_uheapset	umalloc.h	669	Validates all allocated and freed objects on a specified heap, and sets all free memory to a specified value.
_uheap_walk	umalloc.h	671	Returns information about allocated and freed objects on a specified heap.
_ustats	umalloc.h	691	Gets information about a specified heap.

Heap Creation and Management

Function	Header File	Page	Description
_uaddmem	umalloc.h	637	Adds memory to a specified heap.
_uclose	umalloc.h	643	Closes a heap so it can no longer be used.
_ucreate	umalloc.h	646	Creates a heap of memory.
_udefault	umalloc.h	650	Changes the memory heap used as the default heap.
_udestroy	umalloc.h	653	Destroys a heap.
_uopen	umalloc.h	689	Opens a heap so it can be used.

Memory Objects

Function	Header File	Page	Description
memccpy	string.h memory.h	395	Copies a buffer up to and including a specified character or number.
memchr	string.h memory.h	396	Searches a buffer for the first occurrence of a given character.
memcmp	string.h memory.h	397	Compares two buffers.
memcpy	string.h memory.h	399	Copies a buffer.
memicmp	string.h memory.h	400	Compares two buffers without regard to case.
memmove	string.h memory.h	402	Moves a buffer.
memset	string.h memory.h	403	Sets a buffer to a given value.
swab	stdlib.h	608	Copies bytes from a specified source and swaps each pair of adjacent bytes.

Environment Interaction

Function	Header File	Page	Description
abort	stdlib.h process.h	39	Terminates a program abnormally.
exit	stdlib.h process.h	179	Ends a program normally.
getenv	stdlib.h	279	Searches environment variables for a specified variable.
longjmp	setjmp.h	360	Restores a stack environment.
setjmp	setjmp.h	497	Saves a stack environment.
system	stdlib.h process.h	613	Passes a string to the operating system's command interpreter.

Setting and Querying Locale

Function	Header File	Page	Description
csid	stdlib.h	111	Determines the character set identifier for a character.
getsyntx	variant.h	283	Determines the encoding of special characters in the LC_SYNTAX locale category.
iconv	iconv.h	302	Converts characters from one codeset to another.
iconv_close	iconv.h	305	Deletes the conversion descriptor created by iconv_open.
iconv_open	iconv.h	306	Creates a conversion descriptor for iconv to use in converting characters.
localdtconv	locale.h	350	Queries the date and time formatting conventions for the current locale.
localeconv	locale.h	352	Queries the numeric formatting conventions for the current locale.
nl_langinfo	langinfo.h nl_types.h	414	Retrieves requested information for the current locale.
setlocale	locale.h	499	Changes or queries the locale.
wcsid	stdlib.h	721	Determines the character set identifier for a wide character.
wctype	wchar.h	755	Returns the handle for a character class or property.

String Operations

Function	Header File	Page	Description
strcat	string.h	536	Concatenates two strings.
strchr	string.h	537	Locates the first occurrence of a specified character in a string.
strcmp	string.h	538	Compares the value of two strings.
strcmpi	string.h	540	Compares two strings without sensitivity to case.
strcoll	string.h	542	Compares two strings based on the collating elements for the current locale.
strcpy	string.h	544	Copies one string into another.
strcspn	string.h	546	Finds the length of the first substring in a string of characters not in a second string.
strdup	string.h	549	Reserves storage space for the copy of a string.
stricmp	string.h	562	Compares two strings without sensitivity to case.
strlen	string.h	564	Calculates the length of a string.
strncat	string.h	566	Adds a specified length of one string to another string.
strncmp	string.h	568	Compares two strings up to a specified length.
strncpy	string.h	570	Copies a specified length of one string into another.
strnicmp	string.h	572	Compares two strings up to a specified length, without sensitivity to case.
strnset	string.h	574	Sets all characters in a specified length of string to a specified character.
strpbrk	string.h	575	Locates specified characters in a string.
strrchr	string.h	581	Locates the last occurrence of a character within a string.
strrev	string.h	583	Reverses the order of characters in a string.
strset	string.h	574	Sets all characters in a string to a specified character.

Function	Header File	Page	Description
strspn	string.h	585	Locates the first character in a string that is not part of specified set of characters.
strstr	string.h	587	Locates the first occurrence of a string in another string.
strtok	string.h	592	Locates a specified token in a string.
strxfrm	string.h	605	Transforms strings according to locale.

Character Testing

Function	Header File	Page	Description
isalnum	ctype.h	315	Tests for alphanumeric characters.
isalpha	ctype.h	315	Tests for alphabetic characters.
isascii	ctype.h	319	Tests if an integer is within the ASCII range.
isblank	ctype.h	323	Tests for the blank character attribute.
iscntrl	ctype.h	315	Tests for control characters.
_iscsym	ctype.h	325	Tests if a character is alphabetic or an underscore.
_iscsymf	ctype.h	325	Tests if a character is alphabetic, a digit, or an underscore.
isdigit	ctype.h	315	Tests for decimal digits.
isgraph	ctype.h	315	Tests for printable characters excluding the space.
islower	ctype.h	315	Tests for lowercase letters.
isprint	ctype.h	315	Tests for printable characters including the space.
ispunct	ctype.h	315	Tests for printable characters excluding the space.
isspace	ctype.h	315	Tests for white-space characters.
isupper	ctype.h	315	Tests for uppercase letters.
iswalnum	wctype.h	327	Tests for alphanumeric wide characters.
iswalpha	wctype.h	327	Tests for alphabetic wide characters.
iswblank	wctype.h	331	Tests a wide character for the blank character attribute.
iswctype	wctype.h	333	Tests a wide character for a specified property.
iswdigit	wctype.h	327	Tests wide characters for decimal digits.
iswgraph	wctype.h	327	Tests for printable wide characters excluding the space.
iswlower	wctype.h	327	Tests wide characters for lowercase letters.
iswprint	wctype.h	327	Tests for printable wide characters including the space.
iswpunct	wctype.h	327	Tests for printable wide characters excluding the space.
iswspace	wctype.h	327	Tests for white-space wide characters.
iswupper	ctype.h	327	Tests wide characters for uppercase letters.
isxdigit	ctype.h	315	Tests for hexadecimal digits.
iswxdigit	wctype.h	327	Tests wide characters for hexadecimal digits.
wcwidth	wchar.h	758	Determines number of display positions required to display a given wide character.

Character Case Mapping

Function	Header File	Page	Description
strlwr	string.h	565	Converts any uppercase letters in a string to lowercase.
strupr	string.h	604	Converts any lowercase letters in a string to uppercase.
_toascii	ctype.h	631	Converts a value to its ASCII character equivalent.
tolower	ctype.h	631	Converts a character to lowercase.
_tolower	ctype.h	628	Converts an uppercase ASCII character to lowercase.
toupper	ctype.h	631	Converts a character to uppercase.
_toupper	ctype.h	628	Converts a lowercase ASCII character to uppercase.
towlower	wctype.h	632	Converts an uppercase wide character to a lowercase.
towupper	wctype.h	632	Converts a lowercase wide character to uppercase.

Wide Character String Operation Functions

Function	Header File	Page	Description
mblen	stdlib.h	382	Determines length of string.
wcscat	wcstr.h	708	Concatenates **wchar_t** strings.
wcschr	wcstr.h	709	Searches **wchar_t** string for character.
wcscmp	wcstr.h	711	Compares **wchar_t** strings.
wcscoll	wchar.h	713	Compares two wide character strings based on collating elements for the current locale.
wcscpy	wcstr.h	715	Copies **wchar_t** string.
wcscspn	wcstr.h	717	Searches **wchar_t** string for characters.
wcslen	wcstr.h	722	Finds length of **wchar_t** string.
wcsncat	wcstr.h	723	Concatenates **wchar_t** string segment.
wcsncmp	wcstr.h	725	Compares **wchar_t** string segments.
wcsncpy	wcstr.h	727	Copies **wchar_t** string segments.
wcspbrk	wcstr.h	729	Locates **wchar_t** characters in string.
wcsspn	wcstr.h	735	Finds number of **wchar_t** characters.
wcsrchr	wcstr.h	731	Locates **wchar_t** character in string.
wcsstr	wchar.h	736	Locates the first occurrence of a wide character string within another wide character string.
wcstok	wchar.h	739	Locates a specified token in a wide character string.
wcswcs	wcstr.h	747	Locates a **wchar_t** string in another **wchar_t** string.
wcswidth	wchar.h	748	Determines the number of display positions required to display a given wide character string.
wcsxfrm	wchar.h	749	Transforms wide character strings according to the current locale.

Intrinsic Functions

The VisualAge for C++ compiler inlines some functions instead of generating a function call for them. Some of these functions are always inlined; others are inlined only when you compile with the optimization option (/O or /Oc) on.

Functions that Are Inlined when Optimization Is On

When optimization is on (/O+), VisualAge for C++ compiler by default inlines (generates code instead of a function call) the following C library functions:

_clear87	_status87	strncmp	wcslen
_control87	strcat	strncpy	wcsncat
memchr	strchr	strrchr	wcsncmp
memcmp	strcmp	wcscat	wcsncpy
memcpy	strcpy	wcschr	wcsrchr
memmove	strlen	wcscmp	
memset	strncat	wcscpy	

The compiler inlines these functions when you include the appropriate header file that contains the function prototype and the **#define** and **#pragma** statements for the function.

If you program in C, you can override the inlining either by undefining the macro or by placing the name of the function in parentheses, thus disabling the preprocessor substitution. The function then remains a function call, and is not replaced by the code. The size of your object module is reduced, but your application program runs more slowly.

Note: The optimize-for-size compiler option (/Oc) also disables the inlining of non-intrinsic functions. It should be /O+ /Oc-.

Functions that Are Always Inlined

The following functions are built-in functions, meaning they do not have any backing library functions, and are **always** inlined:

abs	_fasin	_fyl2xp1	_llrotr	_srotl
_alloca	_fcos	_f2xm1	_lrotl	_srotr
_crotl	_fcossin	_inp	_lrotr	__sxchg
_crotr	_fpatan	_inpd	__lxchg	
__cxchg	_fptan	_inpw	_outp	
_disable	_fsin	_interrupt	_outpd	
_enable	_fsincos	labs	_outpw	
fabs	_fsqrt	llabs	_rotl	
_facos	_fyl2x	_llrotl	_rotr	

Do not parenthesize the names of these functions.

The built-in functions are all defined in <builtin.h>, in addition to the standard header definitions.

Differentiating between Memory Management Functions

The memory management functions defined by ANSI are calloc, malloc, realloc, and free. These regular functions allocate and free memory from the default runtime heap. (VisualAge for C++ has added another function, _heapmin, to return unused memory to the system.) VisualAge for C++ also provides different versions of each of these functions as extensions to the ANSI definition.

All the versions actually work the same way; they differ only in what heap they allocate from, and in whether they save information to help you debug memory problems. The memory allocated by all of these functions is suitably aligned for storing any type of object.

The following table summarizes the different versions of memory management functions, using malloc as an example of how the names of the functions change for each version. They are all described in greater detail after the table.

	Regular Version	Debug Version
Default Heap	malloc	_debug_malloc
User Heap	_umalloc	_debug_umalloc

To use these extensions, you must set the language level to extended, either with the /Se compiler option or the **#pragma langlvl(extended)** directive.

Heap-Specific Functions

Use the heap-specific versions to allocate and free memory from a user-created heap that you specify. (You can also explicitly use the runtime heap if you want.) Their names are prefixed by _u (for "user heaps"), for example, _umalloc, and they are defined in <umalloc.h>.

The functions provided are:

_ucalloc
_umalloc
_uheapmin

Notice there is no heap-specific version of realloc or free. Because they both always check what heap the memory was allocated from, you can always use the regular versions regardless of what heap the memory came from.

For more information about creating your own heaps and using the heap-specific memory management functions, see Managing Memory with Multiple Heaps in the *Programming Guide*.

Debug Functions

Use these functions to allocate and free memory from the default runtime heap, just as you would use the regular versions. They also provide information that you can use to debug memory problems.

Note: The information provided by these functions is Diagnosis, Modification, and Tuning information only. It is **not** intended to be used as a programming interface.

When you use the debug memory compiler option, /Tm, all calls to the regular memory management functions are mapped to their debug versions. You can also call the debug versions explicitly.

Note: If you parenthesize the calls to the regular memory management functions, they are **not** mapped to their debug versions.

We recommend you place a **#pragma strings(readonly)** directive at the top of each source file that will call debug functions, or in a common header file that each includes. This directive is not essential, but it ensures that the file name passed to the debug functions can't be overwritten, and that only one copy of the file name string is included in the object module.

The names of the debug versions are prefixed by _debug_, for example, _debug_malloc, and they are defined in <malloc.h> and <stdlib.h>.

The functions provided are:

_debug_calloc
_debug_free
_debug_heapmin
_debug_malloc
_debug_realloc

In addition to their usual behavior, these functions also store information (file name and line number) about each call made to them. Each call also automatically checks the heap by calling _heap_check (described below).

Three additional debug memory management functions do not have regular counterparts:

_dump_allocated

Prints information to file standard error handle (the usual destination of **stderr**) about each memory block currently allocated by the debug functions. You can change the destination of the information with the _set_crt_msg_handle function.

_dump_allocated_delta

Prints information to file standard error handle about each memory block allocated by the debug functions since the last call to _dump_allocated or _dump_allocated_delta. Again, you can change the destination of the information with the _set_crt_msg_handle function.

_heap_check

Checks all memory blocks allocated or freed by the debug functions to make sure that no overwriting has occurred outside the bounds of allocated blocks or in a free memory block.

The debug functions call _heap_check automatically; you can also call it explicitly. To use _dump_allocated and _dump_allocated_delta, you must call them explicitly.

In VisualAge C++ releases prior to VisualAge for C++ Version 3.0, you could not mix debug and regular versions of the memory management functions. For example, you could not allocate memory with malloc and free it with _debug_free. This restriction no longer applies; realloc and free (debug or otherwise) can now handle memory allocated by any other allocation function.

Heap-Specific Debug Functions

The heap-specific functions also have debug versions that work just like the regular debug versions. Use these functions to allocate and free memory from the user-created heap you specify, and also provide information that you can use to debug memory problems in your own heaps.

Note: The information provided by these functions is Diagnosis, Modification, and Tuning information only. It is **not** intended to be used as a programming interface.

You can call them explicitly, or you can use the debug memory compiler option, /Tm, to map calls to the heap-specific functions to their debug counterparts.

Note: If you parenthesize the calls to the heap-specific memory management functions, they are **not** mapped to their debug versions.

The names of the heap-specific debug versions are prefixed by _debug_u, for example, _debug_umalloc, and they are defined in <umalloc.h>.

The functions provided are:

_debug_ucalloc
_debug_uheapmin
_debug_umalloc
_udump_allocated
_udump_allocated_delta
_uheap_check

Notice there is no heap-specific debug version of realloc or free. Because they both always check what heap the memory was allocated from, you always use the regular debug versions (_debug_realloc and _debug_free), regardless of what heap the memory came from.

For more information about debugging memory problems in your own heaps, see "Debugging Your Heaps" in the *Programming Guide*.

Infinity and NaN Support

VisualAge for C++ compiler supports the use of infinity and NaN (not-a-number) values. Infinity is a value with an associated sign that is mathematically greater in magnitude than any binary floating-point number. A NaN is a value in floating-point computations that is not interpreted as a mathematical value, and that contains a mask state and a sequence of binary digits.

The value of infinity can be computed from 1.0 / 0.0. The value of a NaN can be computed from 0.0 / 0.0.

Depending on its bit pattern, a NaN can be either quiet (NaNQ) or signalling (NaNS), as defined in the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic* (754-1982). A NaNQ is masked and never generates exceptions. A NaNS may be masked and may generate an exception, but does not necessarily do so. VisualAge for C++ compiler supports only quiet NaN values; all NaN values discussed below refer to quiet NaNs.

NaN and infinity values are defined as macro constants in the <float.h> header file. The macros are:

Macro	Description
_INFINITYF	Infinity of type **float**
_INFINITY	Infinity of type **double**
_INFINITYL	Infinity of type **long double**
_INFF	Same as _INFINITYF
_INF	Same as _INFINITY
_INFL	Same as _INFINITYL
_NANF	Quiet NaN of type **float**
_NAN	Quiet NaN of type **double**
_NANL	Quiet NaN of type **long double**.

You can get the corresponding negative values by using the unary minus operator (for example, -_INF).

Note: The value of 0.0 can also be positive or negative. For example, 1.0 / (-0.0) results in -_INF.

Because these macros are actually references to constant variables, you cannot use them to initialize static variables. For example, the following statements are not allowed:

```
static double infval = _INF;
static float nanval = 1.0 + _NANF;
```

However, you can initialize static variables to the numeric values of infinity and NaN:

```
static double infval = 1.0 / 0.0;
static float nanval =  0.0 / 0.0;
```

Note: Although positive and negative infinities are specific bit patterns, NaNs are not. A NaN value is not equal to itself or to any other value. For example, if you assign a NaN value to a variable x, you cannot check the value of x with the statement if (_NAN == x). Instead, use the statement if (x != x).

All relational and equality expressions involving NaN values always evaluate to FALSE or zero (0), with the exception of not equal (!=), which always evaluates to TRUE or one (1).

For information on the bit mapping and storage mapping of NaN and infinity values, see the *User's Guide*.

Infinity and NaN in Library Functions

When the language level is set to extended (using the /Se option or the **#pragma langlvl(extended)** directive), which is the default, infinity and NaN values can be passed as arguments to the scanf and printf families of library functions, and to the string conversion and math functions. At other language levels, these functions work as described in this book.

This section describes how the library functions handle the infinity and NaN values.

scanf Family

The **scanf** family of functions includes the functions **scanf**, **fscanf**, **sscanf**, and **swscanf**. When reading in floating-point numbers, these functions convert the strings INFINITY, INF, and NAN (in upper-, lower-, or mixed case) to the corresponding floating-point value. The sign of the value is determined by the format specification.

Given a string that consists of NAN, INF, or INFINITY, followed by other characters, the scanf functions read in only the NaN or infinity value, and consider the rest of the string to be a second input field. For example, Nancy would be scanned as two fields, Nan and cy.

Note: In the case of a string that begins with INF, the functions check the fourth letter. If that letter is not I (in upper- or lowercase), INF is read and converted and the rest of the string is left for the next format specification. If

the fourth letter is I, the functions continue to scan for the full INFINITY string. If the string is other than INFINITY, the entire string is discarded.

Example: In the following example, fscanf converts NaN and infinity strings to their numeric values.

```
#include <stdio.h>

int main(void)
{
     int n, count;
     double d1, d2, d3;
    FILE  *stream;

     stream = tmpfile();

     fputs(" INFINITY NAn INF", stream);

    rewind(stream);

     n = fscanf(stream, "%lF%lf%lF%n", &d1, &d2, &d3, &count);

     if (n != EOF)
    {
          printf("Number of fields converted = %d\n", n);
          printf("Number of characters read = %d\n", count);
          printf("Output = %f %F %F\n", d1, d2, d3);
    }

    return  0;

     /* The expected output is:

          Number of fields converted = 3
          Number of characters read = 17
          Output = infinity NAN INFINITY    */

}
```

Figure 1. Example of fscanf *Using NaN and Infinity Values*

For more information on the **scanf**, **fscanf**, **sscanf**, and **swscanf** functions, see the entries for each function in Chapter 3, “Library Functions” on page 37.

Infinity and NaN Support

printf Family

The **printf** family of functions includes the functions **printf**, **fprintf**, **sprintf**, **swprintf**, **vfprintf**, **vprintf**, **vsprintf**, and **vswprintf**. These functions convert floating-point values of infinity and NaN to the strings "INFINITY" or "infinity" and "NAN" or "nan".

The case is determined by the format specification, as is the sign (positive or negative). When converting these values, the **printf** functions ignore the precision width given by the format specification.

Example: In the following example, **printf** converts the NaN and infinity values and prints out the corresponding string.

```
#include <stdio.h>
#include <float.h>

int main(void)
{
    double infval = -(_INF);
    float nanval = _NANF;

    printf("-_INF is the same as %-15.30f\n", infval);
    printf("_NANF is the same as %-15.30F\n", nanval);

   return  0;

    /* The expected output is:

        -_INF is the same as -infinity
        _NANF is the same as NAN              */

}
```

Figure 2. Example of printf *Using NaN and Infinity Values*

For more information on the **printf**, **fprintf**, **sprintf**, **vfprintf**, **vprintf**, and **vsprintf** functions, see the entries for each function in Chapter 3, "Library Functions" on page 37.

String Conversion Functions

The string conversion functions that support infinity and NaN representations include the functions atof, _atold, _ecvt, _fcvt, _gcvt, strtod, strtold, and wcstod.

The atof, _atold, strtod, strtold, and wcstod functions accept the strings INFINITY, INF, and NAN (in upper-, lower-, or mixed case) as input, and convert these strings to the corresponding macro value defined in <float.h>. The _ecvt, _fcvt, and _gcvt functions convert infinity and NaN values to the strings INFINITY and NAN, respectively.

Note: If a signalling NaN string is passed to a string conversion function, a quiet NaN value is returned, and no signal is raised.

Example: The following example uses **atof** to convert the strings "naN" and "inf" to the corresponding macro value.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *nanstr;
   char  *infstr;

    nanstr = "naN";
    printf( "Result of atof = %.10e\n", atof(nanstr) );

    infstr = "inf";
    printf( "Result of atof = %.10E\n", atof(infstr) );

   return  0;

    /* The expected output is:

        Result of atof = nan
        Result of atof = INFINITY    */

}
```

Figure 3. Example of atof *Using NaN and Infinity Values*

For more information on the individual string conversion functions, refer to the entries for them in Chapter 3, “Library Functions” on page 37.

Infinity and NaN Support

Math Functions

Most math functions accept infinity, negative infinity, and NaN values as input. (For information on those functions that do not accept these values, see "Math Functions without Infinity and NaN Support" on page 34.) In general, a NaN value as input results in a NaN value as output, and infinity values as input usually result in infinity values. If the input value is outside the domain or range of the function, errno is set to EDOM or ERANGE, respectively.

The following tables display the results of each math function when NaN or infinity values are input, and the associated errno value if one exists. The first table lists the functions that take only one argument; the second lists those that take two arguments.

Note: In some cases, infinity is always a valid input value for the function regardless of the language level (for example, atan). These cases do not appear in these two tables.

Table 1 (Page 1 of 2). NaN and Infinity Values in Math Functions

Function	Input	Result	errno Value
acos	NaN	NaN	
asin	infinity	0	EDOM
	-infinity	0	EDOM
atan	NaN	NaN	
ceil	NaN	NaN	
floor	infinity	infinity	
	-infinity	-infinity	
cos	NaN	NaN	EDOM
tan	infinity	NaN	ERANGE
	-infinity	NaN	ERANGE
cosh	NaN	NaN	
	infinity	infinity	ERANGE
	-infinity	infinity	ERANGE
erf	NaN	NaN	EDOM
	infinity	1	
	-infinity	-1	
erfc	NaN	NaN	EDOM
	infinity	0	
	-infinity	2	
exp	NaN	NaN	
	infinity	infinity	ERANGE
	-infinity	0	ERANGE
fabs	NaN	NaN	
	infinity	infinity	
	-infinity	infinity	

Table 1 (Page 2 of 2). NaN and Infinity Values in Math Functions

Function	**Input**	**Result**	errno **Value**
frexp	NaN	NaN, 0	EDOM
	infinity	NaN, 0	EDOM
	-infinity	NaN, 0	EDOM
gamma	NaN	NaN	EDOM
	infinity	infinity	ERANGE
	-infinity	NaN	EDOM
log log10	NaN	NaN	
	infinity	infinity	
	0	-infinity	ERANGE
	<0	NaN	EDOM
modf	NaN	NaN, NaN	EDOM
	infinity	NaN, infinity	EDOM
	-infinity	NaN, -infinity	EDOM
sin	NaN	NaN	EDOM
	infinity	NaN	ERANGE
	-infinity	NaN	ERANGE
sinh	NaN	NaN	EDOM
	infinity	infinity	ERANGE
	-infinity	-infinity	ERANGE
sqrt	NaN	NaN	
	infinity	infinity	
	-infinity	0	EDOM
tanh	NaN	NaN	EDOM
	infinity	1	
	-infinity	-1	

Infinity and NaN Support

The functions in the following table take two arguments. The results from NaN and infinity values vary depending on which argument they are passed as.

Table 2. Infinity and NaN Values in Math Functions

Function	Argument 1	Argument 2	Result	errno Value
atan2	NaN	any number	NaN	EDOM
	any number	NaN	NaN	
fmod	NaN	any number	NaN	EDOM
	any number	NaN	NaN	EDOM
	±infinity	any number	0	EDOM
ldexp	infinity	any number	infinity	ERANGE
	-infinity	any number	-infinity	ERANGE
	NaN	any number	NaN	EDOM
pow	±infinity	0	NaN	EDOM
	infinity	-infinity	NaN	EDOM
	-infinity	±infinity	NaN	EDOM
	-infinity	<-1	NaN	EDOM
	-infinity	<1, >-1	NaN	EDOM
	-infinity	>1	NaN	EDOM
	NaN	any number	NaN	EDOM
	any number	NaN	NaN	EDOM
	≤0	infinity	NaN	EDOM
	1	±infinity	NaN	EDOM
	±infinity	±1	0	ERANGE
	>0, <1	infinity	0	ERANGE
	>0, <1	-infinity	infinity	ERANGE

Note: If a signalling NaN is passed to a math function, the behavior is undefined.

Math Functions without Infinity and NaN Support

The following floating-point unit functions are not supported for use with infinity or NaN values. In general, a NaN or infinity value as input for these functions results in an undefined value, an invalid operation exception, or both. These functions do not set errno.

_facos	_fasin	_fcos	_fcossin
_fpatan	_fptan	_fsin	_fsincos
_fsqrt	_fyl2x	_fyl2xp1	_f2xm1

Note: If you expect the return value of a math function to be infinity or NaN, you should use the functions that are supported for these values. The advantage in using the floating-point unit math functions is a reduction in the processing time and in the size of your executable file.

Using Low-Level I/O Functions

The VisualAge for C++ compiler supports both stream and low-level I/O. The primary difference between the two types of I/O is that low-level I/O leaves the responsibility of buffering and formatting up to you.

In general, you should not mix input or output from low-level I/O with that from stream I/O. The only way to communicate between stream I/O and low-level I/O is by using the fdopen or fileno functions.

The low-level I/O functions include:

access	dup2	isatty	stat
chmod	__eof	lseek	_tell
_chsize	fdopen	open	umask
close	_filelength	read	write
creat	fileno	_setmode	
dup	fstat	_sopen	

When you use the low-level I/O functions, you should be aware of the following:

A handle is a value that identifies a file. It is created by the system and used by low-level I/O functions. For VisualAge for C++, the handle returned by low-level I/O functions like open (called the *C_handle*) is the same as that returned by CreateFile (called the *API_handle*). As a result, you can get a file handle using the low-level I/O functions, and then use it with Win32 APIs.

Portability Note: Other compilers may map the file handle so that the *C_handle* and *API_handle* are different. If you will be compiling your programs with other compilers, do not write code that depends on the file handles being the same.

You can pass handles between library environments without restriction. If you acquire a handle other than by using the VisualAge for C++ library functions open, creat, _sopen, or fileno, you must run _setmode for that handle before you use it with other VisualAge for C++ library functions.

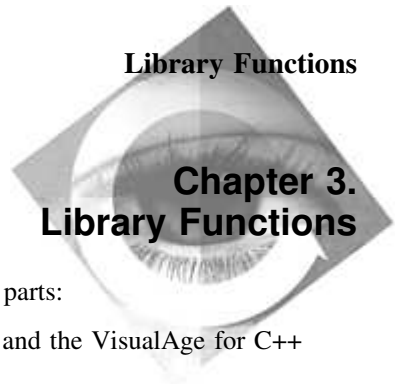
The default open-sharing mode is SH_DENYWR. Use _sopen to obtain other sharing modes.

Text mode deletes '\r' characters on input and changes '\n' to '\r\n' on output.

In a multithread environment, you must ensure that two threads do not attempt to perform low-level I/O operations on the same file at the same time. You must make sure that one I/O process is completed before another begins.

If the file mode is text, the low-level I/O functions treat the character 'x1a' in the following ways:

- If it is detected in a nonseekable file, 'x1a' is treated as end-of-file. In a seekable file, it is treated as end-of-file only if it is the last character.
- If a file is opened as text with either the O_APPEND or O_RDWR flags and 'x1a' is the last character of the file, the last character of the file is deleted.



Chapter 3. Library Functions

The VisualAge for C++ libraries are divided into two parts:

Standard libraries, which define the SAA features and the VisualAge for C++ standard extensions to SAA

Subsystem libraries, which are a subset of the standard libraries, and are used for subsystem development. The functions in these libraries do not require a runtime environment.

This section lists alphabetically and describes all the functions that the VisualAge for C++ product offers, including the extensions to the ANSI/ISO C definition. For information on the subsystem libraries and the functions in them, see the chapter called "Developing Subsystems" in the *Programming Guide*.

Each function description includes the following subsections:

Format
The prototyped declaration of the function and the header file in which it is found. To include the declaration in your code, include the header file.

Description
A brief description of what the function does, what parameters it takes, and how to use the function.

Language Level
The C language standard that each function belongs to (some fall under more than one):

ANSI	ANSI/ISO 9899-1990[1992] C standard (commonly referred to as the ANSI C standard or ANSI/ISO C standard).
ANSI 93	A subset of the ISO/IEC 9899:1990/Amendment 1:1993(E).
POSIX	ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard.
XPG4	X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4.
SAA	IBM Systems Application Architecture Common Programming Interface (SAA CPI) Level 2 definition.
Extension	Extensions to the conventional standards, often specific to VisualAge for C++. (These include standard functions that have additional features under VisualAge for C++.)

Note: To use the library extensions, you must set the language level to extended (with the **`#pragma langlvl(extended)`** directive or the /Se option); note that this is the default.

© Copyright IBM Corp. 1992, 1996

Return Value

The value returned from a successful call to the function and the error return value.

Example

A short example of how to use the function. From the online *C Library Reference*, you can use the IPF **Copy to File** choice from the **Services** pull-down to copy a function example to a separate file (called TEXT.TMP by default). You can then compile, link, and run the example, or use the example code in your own source files.

Related Information

A list of other functions that are similar to or related to the function, and of other topics that provide additional information that help you use the function.

abort — Stop a Program

Format

```
#include <stdlib.h>
void abort(void);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

abort causes an abnormal program termination and returns control to the host environment. It is similar to exit, except that abort does not flush buffers and close open files before ending the program. Calls to abort raise the SIGABRT signal.

Return Value

There is no return value.

This example tests for successful opening of the file myfile.mjq. If an error occurs, an error message is printed and the program ends with a call to abort.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE  *stream;

    if (NULL == (stream = fopen("myfile.mjq", "r"))) {
        perror("Could not open data file");
       abort();
   }
   return  0;

   /*************************************************************************
        If myfile.mjq doesn't exist, the output should be:

        Could not open data file: The file cannot be found.
   *************************************************************************/
}
```

"exit — End Program" on page 179
"_exit — End Process" on page 180
"signal — Handle Interrupt Signals" on page 510
"<stdlib.h>" on page 775

abs — Calculate Integer Absolute Value

Format

#include <stdlib.h>
int abs(int *n*);

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

abs returns the absolute value of an integer argument *n*.

Return Value

There is no error return value. The result is undefined when the absolute value of the argument cannot be represented as an integer. The value of the minimum allowable integer is defined by -INT_MAX in the <limits.h> include file.

This example calculates the absolute value of an integer x and assigns it to y.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int x = -4,y;

    y = abs(x);                                          /* y = 4              */
    printf("abs( %d ) = %d\n", x, y);
   return  0;

   /*************************************************************************
        The output should be:

        abs( -4 ) = 4
   *************************************************************************/
}
```

"_cabs — Calculate Absolute Value of Complex Number" on page 73
"fabs — Calculate Floating-Point Absolute Value" on page 182
"labs — Calculate Absolute Value of Long Integer" on page 339
"<limits.h>" on page 766
"<stdlib.h>" on page 775

access — Determine Access Mode

Format

```
#include <io.h>
int access(char *pathname, int mode);
```

Description

Language Level: POSIX, XPG4, Extension

access determines whether the specified file exists and whether you can get access to it in the given *mode*. Possible values for the *mode* and their meaning in the access call are:

Value	Meaning
06	Check for permission to read from and write to the file.
04	Check for permission to read from the file.
02	Check for permission to write to the file.
00	Check only for the existence of the file.

Note: In earlier releases of VisualAge C++, access began with an underscore (_access). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _access to access for you.

Return Value

access returns 0 if you can get access to the file in the specified *mode*. A return value of –1 shows that the file does not exist or is inaccessible in the given *mode*, and the system sets errno to one of the following values:

Value	Meaning
EACCESS	Access is denied; the permission setting of the file does not allow you to get access to the file in the specified mode.
ENOENT	The system cannot find the file or the path that you specified, or the file name was incorrect.
EINVAL	The mode specified was not valid.
EOS2ERR	The call to the operating system was not successful.

This example checks for the existence of the file sample.dat. If the file does not exist, it is created.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    if (-1 == access("sample.dat", 00)) {
        printf("File sample.dat does not exist.\n");
        printf("Creating  sample.dat.\n");
        system("echo Sample Program > sample.dat");
        if (0 == access("sample.dat", 00))
            printf("File sample.dat created.\n");
    }
    else
        printf("File sample.dat exists.\n");
    return  0;

    /*****************************************************************************
        The output should be:

        File sample.dat does not exist.
        Creating  sample.dat.
        File sample.dat created.
    *****************************************************************************/
}
```

"chmod — Change File Permission Setting" on page 83
"_sopen — Open Shared File" on page 516
"umask — Sets File Mask of Current Process" on page 679
"<io.h>" on page 764

acos — Calculate Arccosine

Format

#include <math.h>
double acos(double *x*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

acos calculates the arccosine of *x*, expressed in radians, in the range 0 to π.

Return Value

acos returns the arccosine of *x*. The value of *x* must be between -1 and 1 inclusive. If *x* is less than -1 or greater than 1, acos sets errno to EDOM and returns 0.

This example prompts for a value for *x*. It prints an error message if *x* is greater than 1 or less than -1; otherwise, it assigns the arccosine of *x* to *y*.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define   MAX                 1.0
#define   MIN                -1.0

int main(void)
{
   double  x,y;

   printf("Enter  x\n");
   scanf("%lf",  &x);

    /* Output error if not in range */
    if (x > MAX)
         printf("Error: %lf too large for acos\n", x);
   else
         if (x < MIN)
              printf("Error: %lf too small for acos\n", x);
         else  {
              y = acos(x);
              printf("acos( %lf ) = %lf\n", x, y);
         }
   return  0;

   /*****************************************************************************
         For the following input: 0.4

         The output should be:

        Enter  x
         acos( 0.400000 ) = 1.159279
   *****************************************************************************/
}
```

"asin — Calculate Arcsine" on page 49

acos

"atan – atan2 — Calculate Arctangent" on page 53
"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_facos — Calculate Arccosine" on page 183
"_fcos — Calculate Cosine" on page 189
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

_alloca — Temporarily Reserve Storage Block

Format

```
#include <stdlib.h>    /* also in <malloc.h> and <builtin.h> */
void *_alloca(size_t size);
```

Description

Language Level: Extension

_alloca is a built-in function that temporarily allocates *size* bytes of storage space from the program's stack. The memory space is automatically freed when the function that called _alloca returns.

Note: _alloca is faster than other allocation functions such as malloc, but it has several limitations:

Because it is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library:
- You cannot take the address of _alloca.
- You cannot parenthesize a call to _alloca. (Parentheses specify a call to the function's backing code, and _alloca has none).

Because _alloca automatically frees storage after the function that calls it returns, you cannot pass the pointer value returned by _alloca as an argument to the free function.

Because _alloca uses automatic storage, programs calling _alloca must not be compiled using the /Gs+ switch. This switch disables stack probes and does not guarantee that enough stack storage will be available. You should use the /Gs- switch, which is the default setting.

Return Value

_alloca returns a pointer to the reserved space. If _alloca cannot reserve the requested space, the program gets an out of stack exception.

This example uses srand to generate five random numbers. The numbers determine how much space _alloca is to allocate for the array barstr. The result is a ragged two-dimensional array of strings.

_alloca

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main(void)
{
    char *fullbar = "1 2 3 4 5";
   int  range,rval;
   char  *barstr[5];
   int  i;

    printf("Bar representation of 5 random numbers ");
    printf("(each generated from 1 to 5):\n\n");

    /* set seed for rand function using time in seconds                    */
   srand((unsigned  int)time(NULL));

    for (i = 0; i < 5; i++) {
        rval = (rand()+1)%5; /* generate random number from 0 to 4          */

        /* for partial copy of fullbar, allocate space on stack for barstr
             from 2 to 10 characters long + one space for a null character */

        barstr[i] =(char *) _alloca((rval *sizeof(char) << 1)+3);
        memset(barstr[i], '\0', (rval *sizeof(char) << 1)+3);

        /* copy random sized portion of fullbar                            */

        strncpy(barstr[i], fullbar, ((rval+1)*2));
        printf("%s\n", barstr[i]);          /* print random length bar     */
   }
   return  0;

   /*************************************************************************
        The output should be similar to :

          Bar representation of 5 random numbers (each generated from 1 to 5):
         1
          1 2 3 4
          1 2 3
          1 2 3 4 5
         1  2
   *************************************************************************/
}
```

"calloc — Reserve and Initialize Storage" on page 75
"free — Release Storage Blocks" on page 238
"malloc — Reserve Storage Block" on page 376
"realloc — Change Reserved Storage Block Size" on page 452
/Gs option in the *User's Guide*
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

asctime — Convert Time to Character String

Format

```
#include <time.h>
char *asctime(const struct tm *time);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

The asctime function converts time, stored as a structure pointed to by *time*, to a character string. You can obtain the *time* value from a call to gmtime or localtime; either returns a pointer to a tm structure defined in <time.h>. See "gmtime — Convert Time" on page 289 for a description of the tm structure fields.

The string result that asctime produces contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

See "printf — Print Formatted Characters" on page 429 for a description of format specifications. The following are examples of the string returned:

```
Sat Jul 16 02:03:55 1994\n\0
```

or

```
Sat Jul 16  2:03:55 1994\n\0
```

The asctime function uses a 24-hour-clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Return Value

The asctime function returns a pointer to the resulting character string. There is no error return value.

Note: asctime, ctime, and other time functions may use a common, statically allocated buffer to hold the return string. Each call to one of these functions may destroy the result of the previous call.

This example polls the system clock and prints a message giving the current time.

asctime

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
   time_t  ltime;

    /* Get the time in seconds */
   time(&ltime);

    /* Convert it to the structure tm */
    newtime = localtime(&ltime);

    /* Print the local time as a string */
    printf("The current date and time are %s", asctime(newtime));
   return  0;

   /*****************************************************************************
        The output should be similar to :

        The current date and time are Fri Jun 28 13:51 1994
   *****************************************************************************/
}
```

"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"strftime — Convert to Formatted Time" on page 557
"time — Determine Current Time" on page 625
"printf — Print Formatted Characters" on page 429
"<time.h>" on page 779

asin — Calculate Arcsine

Format

#include <math.h>
double asin(double *x*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

asin calculates the arcsine of *x*, in the range $-\pi/2$ to $\pi/2$ radians.

Return Value

asin returns the arcsine of *x*. The value of *x* must be between -1 and 1. If *x* is less than -1 or greater than 1, asin sets errno to EDOM, and returns a value of 0.

This example prompts for a value for *x*. It prints an error message if *x* is greater than 1 or less than -1; otherwise, it assigns the arcsine of *x* to *y*.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define   MAX                 1.0
#define   MIN                -1.0

int main(void)
{
   double  x,y;

   printf("Enter  x\n");
   scanf("%lf",  &x);

    /* Output error if not in range                                          */

    if (x > MAX)
         printf("Error: %lf too large for asin\n", x);
   else
         if (x < MIN)

              printf("Error: %lf too small for asin\n", x);
         else  {
              y = asin(x);
              printf("asin( %lf ) = %lf\n", x, y);
         }
   return  0;

   /*****************************************************************************
         For the following input: 0.2

         The output should be:

        Enter  x
         asin( 0.200000 ) = 0.201358
   *****************************************************************************/
}
```

asin

"acos — Calculate Arccosine" on page 43
"atan – atan2 — Calculate Arctangent" on page 53
"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_fasin — Calculate Arcsine" on page 185
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsin — Calculate Sine" on page 251
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

assert — Verify Condition

Format

#include <assert.h>
void assert(int *expression*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

assert prints a diagnostic message to **stderr** and aborts the program if *expression* is false (zero). The diagnostic message has the format:

Assertion failed: *expression*, file *filename*, line *line-number*.

assert takes no action if the *expression* is true (nonzero).

Use assert to identify program logic errors. Choose an *expression* that holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier NDEBUG to remove the assert calls from the program. If you define NDEBUG to any value with a **#define** directive, the C preprocessor expands all assert invocations to **void** expressions. If you use NDEBUG, you must define it before you include <assert.h> in the program.

Return Value

There is no return value.

Note: assert is implemented as a macro. Do not use the **#undef** directive with assert.

In this example, assert tests *string* for a null string and an empty string, and verifies that *length* is positive before processing these arguments.

assert

```
#include <stdio.h>
#include <assert.h>

void analyze(char *string,int length)
{
    assert(string != NULL);                                    /* cannot be NULL      */
    assert(*string != '\0');                                  /* cannot be empty    */
    assert(length > 0);                                        /* must be positive */
   return;
}

int main(void)
{
    char *string = "ABC";
    int length = 3;

   analyze(string,  length);
    printf("The string %s is not null or empty, and has length %d \n", string,
        length);
   return  0;

   /*********************************************************************************
        The output should be:

        The string ABC is not null or empty, and has length 3
   *********************************************************************************/
}
```

"abort — Stop a Program" on page 39
"<assert.h>" on page 761
#define in the *Language Reference*
#undef in the *Language Reference*

atan – atan2 — Calculate Arctangent

Format

```
#include <math.h>
double atan(double x);
double atan2(double y, double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

atan and atan2 calculate the arctangent of *x* and *y*/*x*, respectively.

Return Value

atan returns a value in the range -π/2 to π/2 radians. atan2 returns a value in the range -π to π radians. If both arguments of atan2 are zero, the function sets errno to EDOM, and returns a value of 0.

This example calculates arctangents using the atan and atan2 functions.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  a,b,c,d;

    c = 0.45;
    d = 0.23;
    a = atan(c);
    b = atan2(c, d);
    printf("atan( %lf ) = %lf\n", c, a);
    printf("atan2( %lf, %lf ) = %lf\n", c, d, b);
   return  0;

   /*****************************************************************************
        The output should be:

        atan( 0.450000 ) = 0.422854
        atan2( 0.450000, 0.230000 ) = 1.098299
   *****************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43
"asin — Calculate Arcsine" on page 49
"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_fpatan — Calculate Arctangent" on page 221
"_fptan — Calculate Tangent" on page 226
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

atexit — Record Program Termination Function

Format

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

atexit records a function, pointed to by *func*, that the system calls at normal program termination. For portability, you should use atexit to register up to 32 functions only. The functions are executed in a last-in, first-out order.

If your application has multiple run-time environments, only those functions that are registered in the environment to which the *.exe* file links are executed on termination. Assume, for example, that your application has an *.exe* and a *.dll*, each linking statically to the run-time (that is, compiled with default option -Gd-). Only those registered functions linked from the .exe file are executed on program termination.

Return Value

atexit returns 0 if it is successful, and nonzero if it fails.

This example uses atexit to call the function goodbye at program termination.

```
#include <stdlib.h>
#include <stdio.h>

void _Optlink goodbye(void)
{

    /* This function is called at normal program termination                    */

    printf("The function goodbye was called at program termination\n");
}

int main(void)
{
   int  rc;

    rc = atexit(goodbye);
    if (rc != 0)
        perror("Error in atexit");
   return  0;

   /*************************************************************************
        The output should be:

        The function goodbye was called at program termination
   *************************************************************************/
}
```

"exit — End Program" on page 179
"_exit — End Process" on page 180
"_onexit — Record Termination Function" on page 416

“signal — Handle Interrupt Signals” on page 510
“<stdlib.h>” on page 775

atof — Convert Character String to Float

Format

```
#include <stdlib.h>
double atof(const char *string);
```

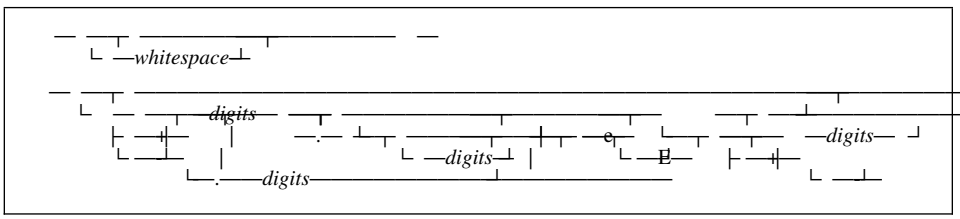
Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

atof converts a character string to a double-precision floating-point value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atof expects a *string* in the following form:



The white space consists of the same characters for which the isspace function is true, such as spaces and tabs. atof ignores leading white-space characters.

For atof, *digits* is one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter e or E. The exponent is a decimal integer, which may be signed.

atof will not fail if a character other than a digit follows an E or if e is read in as an exponent. For example, 100elf will be converted to the floating-point value 100.0. The accuracy is up to 17 significant character digits. The *string* can also be "infinity", "inf", or "nan". These strings are case insensitive, and can be preceded by a unary minus (-). They are converted to infinity and NaN values.

Return Value

atof returns a double value produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

This example shows how to convert numbers stored as strings to numerical values using the atof function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    double  x;
    char  *s;

     s = " -2309.12E-15";
     x = atof(s);                                          /* x = -2309.12E-15 */
     printf("atof( %s ) = %G\n", s, x);
    return  0;

   /**************************************************************************
        The output should be:

        atof(    -2309.12E-15 ) = -2.30912E-12
   **************************************************************************/
}
```

“atoi — Convert Character String to Integer” on page 58
“atol — Convert Character String to Long Integer” on page 60
“_atold — Convert Character String to Long Double” on page 64
“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“Infinity and NaN Support” on page 27
“<stdlib.h>” on page 775

atoi — Convert Character String to Integer

Format

```
#include <stdlib.h>
int atoi(const char *string);
```

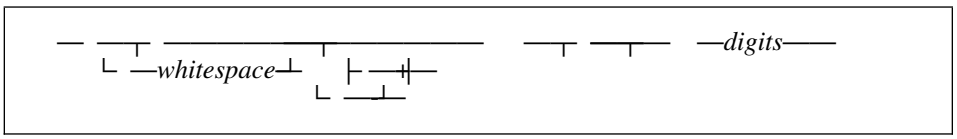
Description

Language Level: ANSI, SAA, POSIX, XPG4

atoi converts a character string to an integer value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atoi does not recognize decimal points nor exponents. The string argument for this function has the form:



where *whitespace* consists of the same characters for which the isspace function is true, such as spaces and tabs. atoi ignores leading white-space characters. *digits* is one or more decimal digits.

Return Value

atoi returns an int value produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. The return value is undefined in the case of an overflow.

This example shows how to convert numbers stored as strings to numerical values using the atoi function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int  i;
    char  *s;

     s = " -9885";
     i = atoi(s);                                                    /* i = -9885          */
     printf("atoi(\"%s\") = %d\n", s, i);
    return  0;

    /*****************************************************************************
         The output should be:

         atoi(" -9885") = -9885
    *****************************************************************************/
}
```

"atof — Convert Character String to Float" on page 56
"atol — Convert Character String to Long Integer" on page 60
"_atold — Convert Character String to Long Double" on page 64
"strtod — Convert Character String to Double" on page 590
"strtol — Convert Character String to Long Integer" on page 594
"strtold — Convert String to Long Double" on page 596
"<stdlib.h>" on page 775

atol — Convert Character String to Long Integer

Format

```
#include <stdlib.h>
long int atol(const char *string);
```

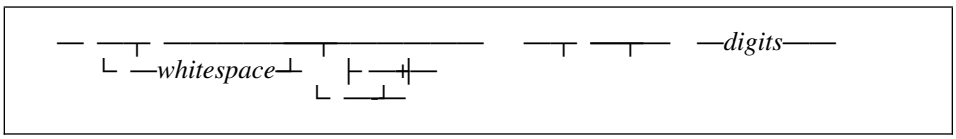
Description

Language Level: ANSI, SAA, POSIX, XPG4

atol converts a character string to a long value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atol does not recognize decimal points nor exponents. The *string* argument for this function has the form:



where *whitespace* consists of the same characters for which the isspace function is true, such as spaces and tabs. atol ignores leading white-space characters. *digits* is one or more decimal digits.

Return Value

atol returns a long value produced by interpreting the input characters as a number. The return value is 0L if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

This example shows how to convert numbers stored as strings to numerical values using the atol function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long  l;
    char  *s;

     s = "98854 dollars";
     l = atol(s);                                                    /* l = 98854            */
     printf("atol( %s ) = %d\n", s, l);
    return  0;

    /*****************************************************************************
         The output should be similar to :

         atol( 98854 dollars ) = 98854
    *****************************************************************************/
}
```

“atof — Convert Character String to Float” on page 56
“atoi — Convert Character String to Integer” on page 58
“_atold — Convert Character String to Long Double” on page 64
“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“<stdlib.h>” on page 775

atoll — Convert Character String to Long Long Integer

Format

```
#include <stdlib.h>
long long int atoll(const char *string);
```

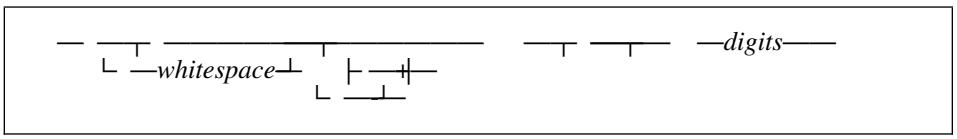
Description

Language Level: Extension

atoll converts a character string to a long long value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atoll does not recognize decimal points nor exponents. The *string* argument for this function has the form:



where *whitespace* consists of the same characters for which the isspace function is true, such as spaces and tabs. atoll ignores leading white-space characters. *digits* is one or more decimal digits.

Return Value

atoll returns a long long value produced by interpreting the input characters as a number. The return value is 0LL if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

This example shows how to convert numbers stored as strings to numerical values using the atoll function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long  ll;
    char  *s;

     s = "98854 dollars";
     ll = atoll(s);                                          /* ll = 98854          */
     printf("atoll( %s ) = %d\n", s, ll);
    return  0;

    /*****************************************************************************
         The output should be similar to :

         atoll( 98854 dollars ) = 98854
    *****************************************************************************/
}
```

“atof — Convert Character String to Float” on page 56
“atoi — Convert Character String to Integer” on page 58
“_atold — Convert Character String to Long Double” on page 64
“atol — Convert Character String to Long Integer” on page 60
“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“<stdlib.h>” on page 775

_atold — Convert Character String to Long Double

Format

```
#include <stdlib.h>       /* also defined in <math.h> */
long double _atold(const char *nptr);
```

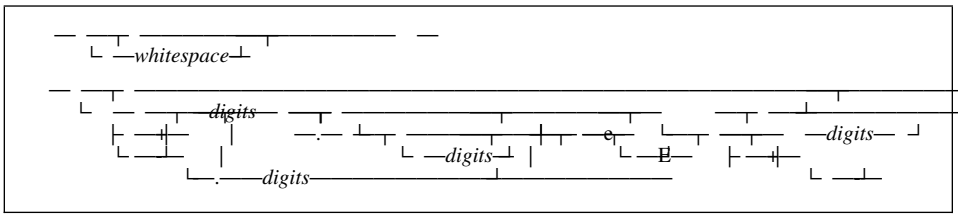
Description

Language Level: Extension

_atold converts a character string pointed to by *nptr* to a long double value. The function continues until it reads a character it does not recognize as part of a number. This character can be the ending null character. Except for its behavior on error, _atold is equivalent to:

```
strtold(nptr, (char **)NULL)
```

The string pointed to by *nptr* must have the following format:



digits is one or more decimal digits. If no digits appear before the decimal point, at least one digit must follow the decimal point. You can place an exponent expressed as a decimal integer after the digits. The exponent can be signed.

The value of *nptr* can also be one of the strings infinity, inf, or nan. These strings are case insensitive, and can be preceded by a unary minus (-). They are converted to infinity and NaN values. For more information on NaN and infinity values, see "Infinity and NaN Support" on page 27.

_atold ignores any white-space characters, as defined by the isspace function.

Return Value

_atold returns the converted long double value. In the case of an underflow, it returns 0. In the case of a positive overflow, _atold returns positive _LHUGE_VAL. It returns negative _LHUGE_VAL for a negative overflow.

This example uses _atold to convert two strings, " -001234.5678e10end of string" and "NaNQ", to their corresponding long double values.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *string;

    string = "    -001234.5678e10end of string";
    printf("_atold = %.10Le\n", _atold(string));
    string = "NaNQ";
    printf("_atold = %.10Le\n", _atold(string));
   return  0;

   /*****************************************************************************
        The output should be:

        _atold = -1.2345678000e+13
        _atold = nan
   *****************************************************************************/

}
```

“atof — Convert Character String to Float” on page 56
“atoi — Convert Character String to Integer” on page 58
“atol — Convert Character String to Long Integer” on page 60
“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“<stdlib.h>” on page 775
“<math.h>” on page 770

_beginthread — Create New Thread

Format

```
#include <stdlib.h>     /* also in <process.h> */
int _beginthread(void (*start_address) (void *),
                 (void  *)stack,
                 unsigned  stack_size,
                 void  *arglist);
```

Description

Language Level: Extension

_beginthread creates a new thread. It takes the following arguments:

start_address This parameter is the address of the function that the newly created thread will execute. When the thread returns from that function, it is terminated automatically. You can also explicitly terminate the thread by calling _endthread.

stack This parameter is ignored. It is retained for compatibility with other compilers.

stack_size The size of the stack, in bytes, that is to be allocated for the new thread. The stack size should be a nonzero multiple of 4K and a minimum of 8K. Memory is used when needed, one page at a time.

arglist A parameter to be passed to the newly created thread. It is the size of a pointer, and is usually the address of a data item to be passed to the new thread, such as a char string. It provides _beginthread with a value to pass to the child thread. NULL can be used as a placeholder.

The function that the new thread will perform **must** be declared and compiled using **_Optlink** linkage.

An alternative to this function is the Win32 CreateThread API. If you use CreateThread, you must also use a #pragma handler statement for the thread function to ensure proper C exception handling. You should also call the _fpreset function at the start of the thread to preset the 387 control status word correctly. When the thread returns from the function, it is terminated automatically. You can also explicitly terminate the thread by calling _endthread.

Note: When using the _beginthread and _endthread functions, you must specify the /Gm+ compiler option to use the multithread libraries.

Return Value If successful, _beginthread returns the thread handle of the new thread. It returns −1 to indicate an error.

This example uses _beginthread to start a new thread bonjour.

Note: To run this example, you must compile it using the /Gm+ compiler option.

```
/*****************************************************************************
Note:    To run this example, you must compile it using
the /Gm+ compiler option.
*****************************************************************************          */

#if    (1 == __TOS_OS2__)
        #define INCL_DOS                                        /* OS/2 */
       #include  <os2.h>
       #define  SLEEP    DosSleep(0l)
#else
        #include <windows.h>                                    /* Windows */
       #define  SLEEP    Sleep(0l)
#endif

#include <stdio.h>
#include <stdlib.h>

static volatile int wait = 1;

void _Optlink bonjour(void *arg)
{
    int i = 0;

    while (wait)                    /* wait until the thread id has been printed        */
          SLEEP;
    while (i++ <5)
       printf("Bonjour!\n");

}

int main(void)
{
   int  tid;

    tid = _beginthread(bonjour, NULL, 8192, NULL);
    if (-1 == tid) {
         printf("Unable to start thread.\n");
        return  EXIT_FAILURE;
    }
```

```
    else  {
        printf("Thread started with thread identifier number %d.\n", tid);
        wait = 0;
    }
#if   (1 == __TOS_OS2__)
   DosWaitThread((PTID)&tid, DCWW_WAIT);   /* wait for thread bonjour to end       */
                                           /* before ending main thread             */
#else
   WaitForSingleObject((HANDLE)tid, INFINITE); /* wait for thread bonjour to end */
                                               /*    before ending main thread        */
#endif

    return  0;

    /*****************************************************************************
        The output should be similar to:

         Thread started with thread identifier number 2.
        Bonjour!
        Bonjour!
        Bonjour!
        Bonjour!
        Bonjour!
    *****************************************************************************/
}
```

"_endthread — Terminate Current Thread" on page 169
"_threadstore — Access Thread-Specific Storage" on page 623
/Gm+ option in the *User's Guide*
"<stdlib.h>" on page 775
"<process.h>" on page 772

Bessel Functions — Solve Differential Equations

Format

```
#include <math.h>
double _j0(double x);
double _j1(double x);
double _jn(int n, double x);
double _y0(double x);
double _y1(double x);
double _yn(int n, double x);
```

Description

Language Level: SAA

Bessel functions solve certain types of differential equations. The _j0, _j1, and _jn functions are Bessel functions of the first kind for orders 0, 1, and *n*, respectively.

The _y0, _y1, and _yn functions are Bessel functions of the second kind for orders 0, 1, and *n*, respectively. The argument *x* must be positive. The argument *n* should be greater than or equal to zero. If *n* is less than zero, it will be a negative exponent.

Return Value

For _j0, _j1, _y0, or _y1, if the absolute value of *x* is too large, the function sets errno to ERANGE, and returns 0. For _y0, _y1, or _yn, if *x* is negative, the function sets errno to EDOM and returns the value -HUGE_VAL. For _y0, _y1, or _yn, if *x* causes an overflow, the function sets errno to ERANGE and returns the value -HUGE_VAL.

This example computes *y* to be the order 0 Bessel function of the first kind for *x*, and *z* to be the order 3 Bessel function of the second kind for *x*.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y,z;

    x = 4.27;
    y = j0(x);          /* y = -0.3660 is the order 0 bessel                    */
                            /* function of the first kind for x                     */
```

```
    printf("j0( 4.27 ) = %5.4f\n", y);
    z = yn(3, x);      /* z = -0.0875 is the order 3 bessel                    */
                       /* function of the second kind for x                    */

    printf("yn( 3,4.27 ) = %5.4f\n", z);
   return  0;

   /*****************************************************************************
        The output should be:

        j0( 4.27 ) = -0.3660
        yn( 3,4.27 ) = -0.0875
   *****************************************************************************/
}
```

"erf – erfc — Calculate Error Functions" on page 173
"gamma — Gamma Function" on page 267
"<math.h>" on page 770

bsearch — Search Arrays

Format

```
#include <stdlib.h>    /* also in <search.h> */
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *key, const void *element));
```

Description

Language Level: ANSI, SAA, POSIX, XPG, Extension

bsearch performs a binary search of an array of *num* elements, each of *size* bytes. The array must be sorted in ascending order by the function pointed to by *compare*. The *base* is a pointer to the base of the array to search, and *key* is the value being sought.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. bsearch calls this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values:

Value	Meaning
Less than 0	*key* less than *element*
0	*key* identical to *element*
Greater than 0	*key* greater than *element*

Return Value

bsearch returns a pointer to *key* in the array to which *base* points. If two keys are equal, the element that *key* will point to is unspecified. If bsearch cannot find the *key*, it returns NULL.

This example performs a binary search on the argv array of pointers to the program parameters and finds the position of the argument PATH. It first removes the program name from argv, and then sorts the array alphabetically before calling bsearch. The functions compare1 and compare2 compare the values pointed to by arg1 and arg2 and return the result to bsearch.

bsearch

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* ------------------------------------------------------------------- */
/* Note: Library always calls functions internally with _Optlink       */
/*       linkage convention.  Ensure that compare1() and compare2      */
/*       are always _Optlink.                                          */
/* ------------------------------------------------------------------- */

int _Optlink compare1(const void *arg1,const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}

int _Optlink compare2(const void *arg1,const void *arg2)
{
    return (strcmp((char *)arg1, *(char **)arg2));
}

int main(int argc,char *argv[])
{
   char  **result;
    char *key = "PATH";
   int  i;
   argv++;
   argc--;

    qsort((char *)argv, argc, sizeof(char *), compare1);
    result = (char **)bsearch(key, argv, argc, sizeof(char *), compare2);
    if (result != NULL) {
        printf("result = <%s>\n", *result);
   }
   else
        printf("result is null\n");
   return  0;

   /*************************************************************************
        If the program name is progname and you enter:

        progname where is PATH in this phrase?

        The output should be:

        result = <PATH>
   *************************************************************************/
}
```

"lfind - lsearch — Find Key in Array" on page 342
"qsort — Sort Array" on page 446
"<stdlib.h>" on page 775

_cabs — Calculate Absolute Value of Complex Number

Format

```
#include <math.h>
double _cabs(struct complex z);
```

Description

Language Level: Extension

_cabs calculates the absolute value of a complex number. This complex number is represented as a structure with the tag *complex* containing the real and imaginary parts. The following type declaration is in <math.h>:

```
struct complex {double x,y;};
```

A call to _cabs is equivalent to:

```
sqrt(z.x * z.x + z.y * z.y)
```

Return Value

_cabs returns the absolute value as a double value. If an overflow results, _cabs calls the _matherr routine and, if necessary, sets **errno** to ERANGE and returns the value HUGE_VAL.

The following program computes the absolute value of the complex number (3.0, 4.0).

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    struct complex value;
   double  d;

    value.x = 3.0;
    value.y = 4.0;
    d = _cabs(value);
    printf("The complex absolute value of %f and %f is %f\n", value.x, value.y, d
        );
   return  0;

   /*****************************************************************************
        The output should be:

        The complex absolute value of 3.000000 and 4.000000 is 5.000000
   *****************************************************************************/
}
```

"abs — Calculate Integer Absolute Value" on page 40
"fabs — Calculate Floating-Point Absolute Value" on page 182
"hypot — Calculate Hypotenuse" on page 301
"labs — Calculate Absolute Value of Long Integer" on page 339

_cabs

"_matherr — Process Math Library Errors" on page 378
"sqrt — Calculate Square Root" on page 527
"<math.h>" on page 770

calloc — Reserve and Initialize Storage

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void *calloc(size_t num, size_t size);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

calloc reserves storage space for an array of *num* elements, each of length *size* bytes. calloc then gives all the bits of each element an initial value of 0.

A heap-specific version of this function (_ucalloc) is also available. calloc always allocates memory from the default heap. You can also use the debug version of calloc, _debug_calloc, to debug memory problems.

Return Value

calloc returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage, or if *num* or *size* is 0.

This example prompts for the number of array entries required and then reserves enough space in storage for the entries. If calloc is successful, the example prints out each entry; otherwise, it prints out an error.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long *array;                        /* start of the array                */
    long *index;                        /* index variable                    */
    int i;                              /* index variable                    */
    int num;                            /* number of entries of the array */
```

```
 printf("Enter the size of the array\n");
scanf("%i",  &num);

 /* allocate num entries                                                    */

 if ((index = array = (long *)calloc(num, sizeof(long))) != NULL) {
      for (i = 0; i < num; ++i)                  /* put values in array          */
           *index++ = i;                              /* using pointer notation          */
      for (i = 0; i < num; ++i)                  /* print the array out          */
           printf("array[ %i ] = %i\n", i, array[i]);
 }
 else {                                                          /* out of storage     */
      perror("Out of storage");
     abort();
 }
 return  0;

 /*************************************************************************
      The output should be similar to :

      Enter the size of the array
     3
      array[ 0 ] = 0
      array[ 1 ] = 1
      array[ 2 ] = 2
 *************************************************************************/
}
```

"_alloca — Temporarily Reserve Storage Block" on page 45
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"free — Release Storage Blocks" on page 238
"malloc — Reserve Storage Block" on page 376
"realloc — Change Reserved Storage Block Size" on page 452
"_ucalloc — Reserve and Initialize Memory from User Heap" on page 641
"<malloc.h>" on page 769
"<stdlib.h>" on page 775
Managing Memory in the *Programming Guide*

ceil — Find Integer >= Argument

Format

```
#include <math.h>
double ceil(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ceil computes the smallest integer that is greater than or equal to *x*.

Return Value

ceil returns the integer as a double value.

This example sets *y* to the smallest integer greater than 1.05, and then to the smallest integer greater than -1.05. The results are 2.0 and -1.0, respectively.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  y,z;

    y = ceil(1.05);                                          /* y = 2.0          */
    printf("ceil( 1.05 ) = %5.f\n", y);
    z = ceil(-1.05);                                         /* z = -1.0         */
    printf("ceil( -1.05 ) = %5.f\n", z);
   return  0;

   /*************************************************************************
        The output should be:

        ceil( 1.05 ) =         2
        ceil( -1.05 ) =       -1
   *************************************************************************/
}
```

"floor — Integer <= Argument" on page 214
"fmod — Calculate Floating-Point Remainder" on page 217
"<math.h>" on page 770

_cgets — Read String of Characters from Keyboard

Format

```
#include <conio.h>
char *_cgets(char *str);
```

Description

Language Level: Extension

_cgets reads a string of characters directly from the keyboard and stores the string and its length in the location pointed to by *str*.

_cgets continues to read characters until it meets a carriage return followed by a line feed (CR-LF) or until it reads the specified number of characters. It stores the string starting at *str*[2]. If _cgets reads a CR-LF combination, it replaces this combination with a null character ('\0') before storing the string. _cgets then stores the actual length of the string in the second array element, *str*[1].

The *str* variable must be a pointer to a character array. The first element of the array, *str*[0], must contain the maximum length, in characters, of the string to be read. The array must have enough elements to hold the string, a final null character, and 2 additional bytes.

Return Value

If successful, _cgets returns a pointer to the actual start of the string, *str*[2]. Otherwise, _cgets returns NULL.

This example creates a buffer and initializes the first byte to the size of the buffer. The program then accepts an input string using _cgets and displays the size and text of that string.

```
#include <conio.h>
#include <stdio.h>

void nothing(void)
{
}

int main(void)
{
    char buffer[82] =    { 84,0 };
   char  *buffer2;
   int  i;

    _cputs("\nPress any key to continue.");
   printf("\n");
    while (0 == _kbhit()) {
        nothing();
   }
   _getch();
    _cputs("\nEnter a line of text:");
   printf("\n");
    buffer2 = _cgets(buffer);
    printf("\nText entered was: %s", buffer2);
   return  0;

   /*****************************************************************************
        The output should be similar to:

        Press any key to continue.

        Enter a line of text:
        This is a simple test.
        Text entered was: This is a simple test.

   *****************************************************************************/
}
```

"_cputs — Write String to Screen" on page 99
"fgets — Read a String" on page 205
"gets — Read a Line" on page 281
"_getch - _getche — Read Character from Keyboard" on page 272
"<conio.h>" on page 762

chdir — Change Current Working Directory

Format

```
#include <direct.h>
int chdir(char *pathname);
```

Description

Language Level: XPG4, Extension

chdir causes the current working directory to change to the directory specified by *pathname*. The *pathname* must refer to an existing directory.

Note: This function can change the current working directory on any drive. It cannot change the default drive. For example, if A:\BIN is the current working directory and A: is the default drive, the following changes only the current working directory on drive C:.

```
chdir ("c:\\emp");
```

A: is still the default drive.

An alternative to this function is the SetCurrentDir API call. For more information, see *Win32 Programmer's Reference*.

Note: In earlier releases of VisualAge C++, chdir began with an underscore (_chdir). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _chdir to chdir for you.

Return Value

chdir returns a value of 0 if the working directory was successfully changed. A return value of -1 indicates an error; chdir sets **errno** to ENOENT, showing that chdir cannot find the specified path name. No error occurs if *pathname* specifies the current working directory.

This example changes the current working directory to the root directory, and then to the \red\green\blue directory.

```
#include <direct.h>
#include <stdio.h>

int main(void)
{
    printf("Changing to the root directory.\n");
   if  (chdir("\\"))
        perror(NULL);
   else
        printf("Changed to the root directory.\n\n");
    printf("Changing to directory '\\red\\green\\blue'.\n");
   if  (chdir("\\red\\green\\blue"))
        perror(NULL);
   else
        printf("Changed to directory '\\red\\green\\blue'.\n");
   return  0;

   /*************************************************************************
        If directory \red\green\blue exists, the output should be:

        Changing to the root directory.
        Changed to the root directory.

        Changing to directory '\red\green\blue'.
        Changed to directory '\red\green\blue'.
   *************************************************************************/
}
```

"_chdrive — Change Current Working Drive" on page 82
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"_getdrive — Get Current Working Drive" on page 278
"system — Invoke the Command Processor" on page 613
"mkdir — Create New Directory" on page 407
"rmdir — Remove Directory" on page 467
"system — Invoke the Command Processor" on page 613
"<direct.h>" on page 762

_chdrive — Change Current Working Drive

Format

```
#include <direct.h>
int _chdrive(int drive);
```

Description

Language Level: Extension

_chdrive changes the current working drive to the *drive* specified. The *drive* variable is an integer value representing the number of the new working drive (A: is 1, B: is 2, and so on).

An alternative to this function is the Win32 SetCurrentDirectory system call. For more information, see *Win32 Programmer's Reference*.

Return Value

_chdrive returns 0 if it is successful in changing the working drive. A return value of -1 indicates an error; _chdrive sets errno to EOS2ERR.

This example uses _chdrive to change the current working drive to C:.

```
#include <direct.h>
#include <stdio.h>

int main(void)
{
   if  (_chdrive(3))
        printf("Cannot change current working drive to 'C' drive.\n");
   else  {
        printf("Current working drive changed to ");
        printf("'%c' drive.\n", ('A'+_getdrive()-1));
   }
   return  0;

   /*************************************************************************
        The output should be similar to :

        Current working drive changed to 'C' drive.
   *************************************************************************/
}
```

"chdir — Change Current Working Directory" on page 80
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"_getdrive — Get Current Working Drive" on page 278
"mkdir — Create New Directory" on page 407
"rmdir — Remove Directory" on page 467
"<direct.h>" on page 762

chmod — Change File Permission Setting

Format

```
#include <io.h>
#include <sys\stat.h>
int chmod(char *pathname, int pmode);
```

Description

Language Level: XPG4, Extension

chmod changes the permission setting of the file specified by *pathname*. The permission setting controls access to the file for reading or writing. You can use chmod only if the file is closed.

The *pmode* expression contains one or both of the constants S_IWRITE and S_IREAD, defined in <sys\stat.h>. The meanings of the values of *pmode* are:

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If you do not give permission to write to the file, chmod makes the file read-only. With the Windows operating system, all files are readable; you cannot give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE set the same permission.

Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

Note: In earlier releases of VisualAge C++, chmod began with an underscore (_chmod). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _chmod to chmod for you.

Return Value

chmod returns the value 0 if it successfully changes the permission setting. A return value of –1 shows an error; chmod sets errno to one of the following values:

Value	Meaning
ENOENT	The system cannot find the file or the path that you specified, or the file name was incorrect.
EOS2ERR	The call to the operating system was not successful.
EINVAL	The mode specified was not valid.

This example opens the file chmod.dat for writing after checking the file to see if writing is permissible. It then writes from the buffer to the opened file. This program takes file names passed as arguments and sets each to read-only.

chmod

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    if (-1 == access("chmod.dat", 00))          /* Check if file exists.          */
        {
        printf("\nCreating  chmod.dat.\n");
         system("echo Sample Program > chmod.dat");
         printf("chmod chmod.dat to be readonly.\n");
         if (-1 == chmod("chmod.dat", S_IREAD))
             perror("Chmod  failed");
         if (-1 == access("chmod.dat", 02))
              printf("File chmod.dat is now readonly.\n\n");
         printf("Run this program again to erase chmod.dat.\n\n");
   }
   else  {
         printf("\nFile chmod.dat exist.\n");
         printf("chmod chmod.dat to become writable.\n");
         if (-1 == chmod("chmod.dat", S_IWRITE))
             perror("Chmod  failed");
        system("erase    chmod.dat");
         printf("File chmod.dat removed.\n\n");
   }
   return  0;

   /*************************************************************************
         If chmod.dat does not exist, the output should be :

          Creating  chmod.dat.
           chmod chmod.dat to be readonly.
           File chmod.dat is now readonly.

           Run this program again to erase chmod.dat.
   *************************************************************************/
}
```


"access — Determine Access Mode" on page 41
"_sopen — Open Shared File" on page 516
"umask — Sets File Mask of Current Process" on page 679
"<sys\stat.h>" on page 778
"<io.h>" on page 764

_chsize — Alter Length of File

Format

#include <io.h>
int _chsize(int *handle*, long *size*);

Description

Language Level: Extension

_chsize lengthens or cuts off the file associated with *handle* to the length specified by *size*. You must open the file in a mode that permits writing. _chsize adds null characters (\0) when it lengthens the file. When _chsize cuts off the file, it erases all data from the end of the shortened file to the end of the original file.

Return Value

_chsize returns the value 0 if it successfully changes the file size. A return value of –1 shows an error; _chsize sets errno to one of the following values:

Value	Meaning
EACCESS	The specified file is locked against access.
EBADF	The file handle is not valid, or the file is not open for writing.
ENOSPC	There is no space left on the device.
EOS2ERR	The call to the operating system was not successful.

This example opens a file named sample.dat and returns the current length of that file. It then alters the size of sample.dat and returns the new length of that file.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
   long  length;
   int  fh;

   printf("\nCreating  sample.dat.\n");
    system("echo Sample Program > sample.dat");
    if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
        printf("Unable to open sample.dat.\n");
       return  EXIT_FAILURE;
   }
```

```
    if (-1 == (length = _filelength(fh))) {
         printf("Unable to determine length of sample.dat.\n");
        return  EXIT_FAILURE;
    }
    printf("Current length of sample.dat is %d.\n", length);
    printf("Changing the length of sample.dat to 20.\n");
    if (-1 == (_chsize(fh, 20))) {
        perror("_chsize  failed");
        return  EXIT_FAILURE;
    }
    if (-1 == (length = _filelength(fh))) {
         printf("Unable to determine length of sample.dat.\n");
        return  EXIT_FAILURE;
    }
    printf("New length of sample.dat is %d.\n", length);
   close(fh);
   return  0;

   /*************************************************************************
         The output should be similar to :

         Creating  sample.dat.
          Current length of sample.dat is 17.
          Changing the length of sample.dat to 20.
          New length of sample.dat is 20.
   *************************************************************************/
}
```

"_filelength — Determine File Length" on page 211
"lseek — Move File Pointer" on page 365
"<io.h>" on page 764

clearerr — Reset Error Indicators

Format

```
#include <stdio.h>
void clearerr (FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

The clearerr function resets the error indicator and end-of-file indicator for the specified *stream*. Once set, the indicators for a specified stream remain set until your program calls clearerr or rewind. fseek also clears the end-of-file indicator.

Return Value

There is no return value.

This example reads a data stream and then checks that a read error has not occurred.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

int main(void)
{
    if (NULL != (stream = fopen("file.dat", "r"))) {
        if (EOF == (c = getc(stream))) {
            if (feof(stream)) {
                perror("Read  error");
                clearerr(stream);
            }
        }
    }
    return  0;

    /*****************************************************************************
        If file.dat is an empty file, the output should be:

        Read error: Attempted to read past end-of-file.

    *****************************************************************************/
}
```

"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"perror — Print Error Message" on page 427
"rewind — Adjust Current File Position" on page 465
"strerror — Set Pointer to Runtime Error Message" on page 550
"_strerror — Set Pointer to System Error String" on page 551
"<stdio.h>" on page 774

_clear87 — Clear Floating-Point Status Word

Format

```
#include <float.h>    /* also in <builtin.h> */
unsigned int _clear87(void);
```

Description

Language Level: Extension

_clear87 gets the floating-point status word and then clears it. The floating-point status word is a combination of the numeric coprocessor status word and other conditions that the numeric exception handler detects, such as floating-point stack overflow and underflow.

_clear87 affects only the current thread. It does not affect any other threads that may be processing.

Return Value

The bits in the value returned reflect the floating-point status before the call to _clear87 was made.

This example takes a number close to 0 as a double and assigns it to a float. The result is a loss of significance, y becomes a denormal number, and the underflow bit of the floating-point status word is set. _clear87 gets the current floating-point status word and then clears it, and printf prints it as immediate data. The result shows the change in the floating-point word because of the loss of significance.

The program then assigns the denormal y to another variable, causing the denormal bit to be set in the floating-point status word. Again, _clear87 gets the current status word and clears it, and printf prints it to the screen.

```
#include <stdio.h>
#include <float.h>

double a = 1e-40;
double b;
float y;
```

```
int main(void)
{
    unsigned int statword;
    unsigned int old_cw;

    /*    change control word to mask all exceptions                    */

   _control87(0x037f,  0xffff);

    /* Assignment of the double to the float y is inexact;              */
    /* the underflow bit is set.                                         */

    y = a;
    statword = _clear87();
    printf("floating-point status = 0x%.4x after underflow\n", statword);
    statword = _status87();
    printf("cleared floating-point status word = 0x%.4x\n", statword);

    /*    reset floating point status word                              */

   _fpreset();

    /*    change control word to mask all exception                     */

   _control87(0x037f,  0xffff);

    /* Reassigning the denormal y to the double b                       */
   /*     causes  the  denormal  bit  to  be  set.                        */

    b = y;
    statword = _clear87();
    printf("floating-point status = 0x%.4x for denormal\n", statword);

    /*    reset floating point status word                              */

   _fpreset();
   return  0;

   /*************************************************************************
        The output should be:

        floating-point status = 0x0030 after underflow
        cleared floating-point status word = 0x0000
        floating-point status = 0x0002 for denormal
   *************************************************************************/
}
```

"_control87 — Set Floating-Point Control Word" on page 94
"_status87 — Get Floating-Point Status Word" on page 534
"_fpreset — Reset Floating-Point Unit" on page 222
"<float.h>" on page 763

clock — Determine Processor Time

Format

```
#include <time.h>
clock_t clock(void);
```

Description

Language Level: ANSI, SAA, XPG4

clock returns an approximation of the processor time used by the program since the beginning of an implementation-defined time-period that is related to the program invocation. To obtain the time in seconds, divide the value returned by clock by the value of the macro CLOCKS_PER_SEC.

Return Value

If the value of the processor time is not available or cannot be represented, clock returns the value (clock_t)-1.

To measure the time spent in a program, call clock at the start of the program, and subtract its return value from the value returned by subsequent calls to clock.

This example prints the time elapsed since the program was invoked.

```
#include <time.h>
#include <stdio.h>

double time1,time2,timedif;                    /* use doubles to show small values */
int i;

int main(void)
{
    time1 = (double)clock();                   /* get initial time in seconds      */
    time1 = time1/CLOCKS_PER_SEC;

    /* use some CPU time                                                            */

    for (i = 0; i < 5000000; i++) {
        int  j;

        j = i;
    }
    time2 = (double)clock();                   /* call clock a second time         */
    time2 = time2/CLOCKS_PER_SEC;
    timedif = time2-time1;
    printf("The elapsed time is %f seconds\n", timedif);
   return  0;

   /*****************************************************************************
        The output should be similar to:

        The elapsed time is 0.969000 seconds
   *****************************************************************************/
}
```

"difftime — Compute Time Difference" on page 140

“time — Determine Current Time” on page 625
“<time.h>” on page 779

close — Close File Associated with Handle

Format

```
#include <io.h>
int close(int handle);
```

Description

Language Level: XPG4, Extension

close closes the file associated with the handle. Use close if you opened the handle with open. If you opened the handle with fopen, use fclose to close it.

Note: In earlier releases of VisualAge C++, close began with an underscore (_close). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _close to close for you.

Return Value

close returns 0 if it successfully closes the file. A return value of −1 shows an error, and close sets errno to EBADF, showing an incorrect file handle argument.

This example opens the file edclose.dat and then closes it using the close function.

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

int main(void)
{
    int  fh;

    printf("\nCreating  edclose.dat.\n");
     if (-1 == (fh = open("edclose.dat", O_RDWR|O_CREAT|O_TRUNC, S_IREAD|S_IWRITE)
         )) {
          perror("Unable to open edclose.dat");
         return  EXIT_FAILURE;
    }
     printf("File was successfully opened.\n");
     if (-1 == close(fh)) {
          perror("Unable to close edclose.dat");
         return  EXIT_FAILURE;
    }
     printf("File was successfully closed.\n");
    return  0;

    /*********************************************************************
          The output should be:

         Creating  edclose.dat.
          File was successfully opened.
          File was successfully closed.
    *********************************************************************/
}
```

"fclose — Close Stream" on page 187
"creat — Create New File" on page 100

"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"<io.h>" on page 764

_control87 — Set Floating-Point Control Word

Format

#include <float.h> /* also in <builtin.h> */
unsigned int _control87(unsigned int *new*, unsigned int *mask*);

Description

Language Level: Extension

_control87 gets the current floating-point control word and then sets it. The floating-point control word specifies the precision, rounding, and infinity modes of the floating-point chip.

You can mask or unmask current floating-point exceptions using _control87. If the value for the mask is equal to 0, _control87 gets the floating-point control word. If the mask is nonzero, _control87 sets a new value for the control word in the manner described below, and returns the previous value of the control word. For any bit in the mask equal to 1, the corresponding bit in *new* updates the control word. This is equivalent to the expression:

fpcntrl = ((fpcntrl & ~*mask*) | (*new* & *mask*))

where *fpcntrl* is the floating-point control word.

_control87 is used for the current thread only. It does not affect any other threads that may be processing.

Warning: If you change the content of the floating-point control word:

- The behavior of the math functions with regard to domain and range errors may be undefined.
- Math functions may not handle infinity and NaN values correctly.
- Some floating-point exceptions may not occur, while other new ones may occur.
- Resetting the EM_INEXACT bit may cause SIG_FPE exceptions, which decrease performance.
- If the precision or rounding bits are modified, you can reduce the precision available for float and double variables.

For information on bits in the control word and handling floating-point exceptions, see the *User's Guide*.

Return Value

The bits in the returned value reflect the floating-point control word before the call.

This example prints the initial control word in hexadecimal, and then illustrates different representations of 0.1, depending on the precision.

```
#include <stdio.h>
#include <float.h>

double a = .13;

int main(void)
{
    printf("control = 0x%.4x\n", _control87(CW_DEFAULT, 0));/* Get control word*/
    printf("a*a = .0169 = %.15e\n", a *a);
    _control87(PC_24, MCW_PC);                  /* Set precision to 24 bits                */
    printf("a*a = .0169 (rounded to 24 bits) = %.15e\n", a *a);
    _control87(CW_DEFAULT, 0xffff);             /* Restore to initial default    */
    printf("a*a = .0169 = %.15e\n", a *a);
   return  0;

   /*************************************************************************
        The output should be similar to:

         control = 0x0362
         a*a = .0169 = 1.690000000000000e-02
         a*a = .0169 (rounded to 24 bits) = 1.690000057220459e-02
         a*a = .0169 = 1.690000000000000e-02
   *************************************************************************/
}
```

"_clear87 — Clear Floating-Point Status Word" on page 88
"_status87 — Get Floating-Point Status Word" on page 534
"_fpreset — Reset Floating-Point Unit" on page 222
Floating Point Variables
"signal — Handle Interrupt Signals" on page 510
"<float.h>" on page 763

cos — Calculate Cosine

Format

```
#include <math.h>
double cos(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

cos calculates the cosine of *x*. The value *x* is expressed in radians. If *x* is too large, a partial loss of significance in the result may occur.

Return Value

cos returns the cosine of *x*.

This example calculates *y* to be the cosine of *x*.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y;

    x = 7.2;
    y = cos(x);
    printf("cos( %lf ) = %lf\n", x, y);
   return  0;

   /*************************************************************************
        The output should be:

        cos( 7.200000 ) = 0.608351
   *************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43
"cosh — Calculate Hyperbolic Cosine" on page 97
"_facos — Calculate Arccosine" on page 183
"_fcos — Calculate Cosine" on page 189
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

cosh — Calculate Hyperbolic Cosine

Format

```
#include <math.h>
double cosh(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

cosh calculates the hyperbolic cosine of *x*. The value *x* is expressed in radians.

Return Value

cosh returns the hyperbolic cosine of *x*. If the result is too large, cosh returns the value HUGE_VAL and sets errno to ERANGE.

This example calculates *y* to be the hyperbolic cosine of *x*.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y;

    x = 7.2;
    y = cosh(x);
    printf("cosh( %lf ) = %lf\n", x, y);
   return  0;

   /*****************************************************************************
       The output should be:

       cosh( 7.200000 ) = 669.715755
   *****************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43
"cos — Calculate Cosine" on page 96
"_facos — Calculate Arccosine" on page 183
"_fcos — Calculate Cosine" on page 189
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

_cprintf — Print Characters to Screen

Format

```
#include <conio.h>
int _cprintf(char *format-string, argument-list);
```

Description

Language Level: Extension

_cprintf formats and sends a series of characters and values directly to the screen, using the _putch function to send each character.

The *format-string* has the same form and function as the *format-string* parameter for printf. Format specifications in the *format-string* determine the output format for any *argument-list* that follows. See "printf — Print Formatted Characters" on page 429 for a description of the *format-string*.

Note: Unlike the fprintf, printf, and sprintf functions, _cprintf does not translate line feed characters into output of a carriage return followed by a line feed.

Return Value

_cprintf returns the number of characters printed.

The following program uses _cprintf to write strings to the screen.

```
#include <conio.h>

int main(void)
{
   char  buffer[24];

    _cprintf("\nPlease enter a filename:\n");
   _cscanf("%23s",  buffer);
    _cprintf("\nThe file name you entered was %23s.", buffer);
   return  0;

   /*****************************************************************************
        The output should be similar to :

         Please enter a filename:
                                    file.dat
         The filename you entered was                    file.dat.
   *****************************************************************************/
}
```

"_cscanf — Read Data from Keyboard" on page 112
"fprintf — Write Formatted Data to a Stream" on page 224
"printf — Print Formatted Characters" on page 429
"_putch — Write Character to Screen" on page 438
"sprintf — Print Formatted Data to Buffer" on page 525
"<conio.h>" on page 762

_cputs — Write String to Screen

Format

```
#include <conio.h>
int _cputs(char *str);
```

Description

Language Level: Extension

_cputs writes the string that *str* points to directly to the screen. The string *str* must end with a null character (\0). The _cputs function does not automatically add a carriage return followed by a line feed to the string.

Return Value

If successful, _cputs returns 0. Otherwise, it returns a nonzero value.

This example outputs a prompt to the screen.

```
#include <conio.h>

int main(void)
{
    char *buffer = "Insert data disk in drive a: \r\n";

   _cputs(buffer);
   return  0;

   /*************************************************************************
        The output should be:

        Insert data disk in drive a:
   *************************************************************************/
}
```

"_cgets — Read String of Characters from Keyboard" on page 78
"fputs — Write String" on page 230
"_putch — Write Character to Screen" on page 438
"puts — Write a String" on page 441
"<conio.h>" on page 762

creat — Create New File

Format

```
#include <io.h>
#include <sys\stat.h>
int creat(char *pathname, int pmode);
```

Description

Language Level: XPG4, Extension

creat either creates a new file or opens and truncates an existing file. If the file specified by *pathname* does not exist, creat creates a new file with the given permission setting and opens it for writing in text mode. If the file already exists, and the read-only attribute and sharing permissions allow writing, creat truncates the file to length 0. This action destroys the previous contents of the file and opens it for writing in text mode. creat always opens a file in text mode for reading and writing.

The permission setting *pmode* applies to newly created files only. The new file receives the specified permission setting after you close it for the first time. The *pmode* integer expression contains one or both of the constants S_IWRITE and S_IREAD, defined in <sys\stat.h>. The values of the *pmode* argument and their meanings are:

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If you do not give permission to write to the file, it is a read-only file. You cannot give write-only permission, thus the modes S_IWRITE and S_IREAD | S_IWRITE have the same results. The creat function applies the current file permission mask to *pmode* before setting the permissions. (See "umask — Sets File Mask of Current Process" on page 679 for more information about file permission masks.)

When writing new code, you should use open rather than creat.

Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

Note: In earlier releases of VisualAge C++, creat began with an underscore (_creat). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _creat to creat for you.

Return Value creat returns a handle for the created file if the call is successful. A return value of –1 shows an error, and creat sets errno to one of the following values:

Value	Meaning
EACCESS	File sharing violated.
EINVAL	The mode specified was not valid.
EMFILE	No more file handles are available.
ENOENT	The path name was not found, or the file name was incorrect.
EOS2ERR	The call to the operating system was not successful.

This example creates the file sample.dat so it can be read from and written to.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int  fh;

    fh = creat("sample.dat", S_IREAD|S_IWRITE);
    if (-1 == fh) {
        perror("Error in creating sample.dat");
       return  EXIT_FAILURE;
   }
   else
        printf("Successfully created sample.dat.\n");
   close(fh);
   return  0;

   /*****************************************************************************
        The output should be:

         Successfully created sample.dat.
   *****************************************************************************/
}
```

"chmod — Change File Permission Setting" on page 83
"close — Close File Associated with Handle" on page 92
"open — Open File" on page 418
"fdopen — Associates Input Or Output With File" on page 194
"_sopen — Open Shared File" on page 516
"umask — Sets File Mask of Current Process" on page 679
"<sys\stat.h>" on page 778
"<io.h>" on page 764

_crotl – _crotr — Rotate Bits of Character Value

Format

```
#include <stdlib.h>       /* also in <builtin.h> */
unsigned char _crotl(unsigned char value, int shift);
unsigned char _crotr(unsigned char value, int shift);
```

Description

Language Level: Extension

The _crotl and _crotr functions rotate the character *value* by *shift* bits. The _crotl function rotates to the left, and _crotr to the right.

Note: Both _crotl and _crotr are built-in functions, which means they are implemented as inline instructions and have no backing code in the library. For this reason:

You cannot take the address of these functions.
You cannot parenthesize a call to either function. (Parentheses specify a call to the function's backing code, and these functions have none.)

Return Value

Both functions return the rotated value. There is no error return value.

This example uses _crotl and _crotr with different shift values to rotate the character value:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned char val = 0X01;

    printf("The value of 0x%2.2x rotated 4 bits to the left is 0x%2.2x\n", val,
        _crotl(val,  4));
    printf("The value of 0x%2.2x rotated 2 bits to the right is 0x%2.2x\n",
        val, _crotr(val, 2));
   return  0;

   /*************************************************************************
        The output should be:

        The value of 0x01 rotated 4 bits to the left is 0x10
        The value of 0x01 rotated 2 bits to the right is 0x40
   *************************************************************************/
}
```

"_lrotl - _lrotr — Rotate Bits of Unsigned Long Value" on page 364
"_rotl - _rotr — Rotate Bits of Unsigned Integer" on page 483
"_srotl - _srotr — Rotate Bits of Unsigned Short Value" on page 529
"<stdlib.h>" on page 775
"<builtin.h>" on page 761

_CRT_init — Initialize DLL Runtime Environment

Format

```
int _CRT_init(void);
/* no header file - defined in the runtime startup code */
```

Description

Language Level: Extension

_CRT_init initializes the VisualAge for C++ runtime environment for a DLL.

By default, all DLLs call the VisualAge for C++ _DLL_InitTerm function, which in turn calls _CRT_init for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than runtime initialization and termination), you must call _CRT_init from your version of _DLL_InitTerm before you can call any other runtime functions to insure that the runtime library is initialized. If _CRT_init has been called, there must be a matching _CRT_term to insure library termination.

If your DLL contains C++ code, you must also call __ctordtorInit after _CRT_init to ensure that static constructors and destructors are initialized properly. The prototype for __ctordtorInit is

```
void __ctordtorInit(int flag)
```

Value 0 for *flag* means that destructors are called for process termination. Value 1 for *flag* means that only destructors for thread specific objects in the current thread should be called.

Note: If you are providing your own version of the _matherr function to be used in your DLL, you must also call the _exception_dllinit function after the runtime environment is initialized. Calling this function ensures that the proper _matherr function will be called during exception handling. _exception_dllinit is defined in the runtime startup code as:

```
void _Optlink _exception_dllinit( int (*)(struct exception *) );
```

The parameter required is the address of your _matherr function.

Return Value

If the runtime environment is successfully initialized, _CRT_init returns 0. A return code of -1 indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of **stderr**.

_CRT_init

This example shows the _DLL_InitTerm function from the VisualAge for C++ sample program for building DLLs, which calls _CRT_init to initialize the library environment.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define   CCHMAXPATH   255
/* _CRT_init is the C run-time environment initialization function.          */
/* It will return 0 to indicate success and -1 to indicate failure.          */

int _CRT_init(void);
#ifdef     STATIC_LINK

/* _CRT_term is the C run-time environment termination function.             */
/* It only needs to be called when the C run-time functions are statically   */
/* linked.                                                                   */
void _CRT_term(void);
#else


static void    cleanup(void);
#endif
size_t nSize;
int *pArray;

/* _DLL_InitTerm is the function that gets called by the operating system    */
/* loader when it loads and frees this DLL for each process that accesses    */
/* this DLL.    However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.    The system      */
/* linkage convention MUST be used because the operating system loader is    */
/* calling this function.                                                    */

unsigned long _System _DLL_InitTerm(unsigned long hModule, unsigned long
                                    ulFlag, long * dummy)
{
    size_t  i;
    int    rc;
    char  namebuffflCCHMAXPATH";
```

```
    /* If ulFlag is one then the DLL is being loaded so initialization should*/
    /* be performed.    If ulFlag is zero then the DLL is being freed so          */
    /* termination should be performed.                                          */

    switch (ulFlag) {
        case 1 :

            /*************************************************************************/
             /* The C run-time environment initialization function must be          */
             /* called before any calls to C run-time functions that are not       */
             /*  inlined.                                                           */
            /*************************************************************************/

             if (_CRT_init() == -1)
                 return  0UL;
#ifndef   STATIC_LINK

            /*************************************************************************/
             /* Cleanup routine is registered by atexit function to clean up       */
             /* if the runtime is dynamically linked.                              */
            /*************************************************************************/

                 if  (atexit(cleanup))
                      printf("atexit for cleanup failed\n");
#endif
             if ((rc = GetModuleFileName((void *)hModule, namebuf, CCHMAXPATH)) == 0)
                  printf("GetModuleFileName returned %lu\n", GetLastError());
            else
                  printf("The name of this DLL is %s\n", namebuf);
            srand(17);
             nSize = (rand()%128)+32;
             printf("The array size for this process is %u\n", nSize);
             if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
                  printf("Could not allocate space for unsorted array.\n");
                 return  0UL;
            }
             for (i = 0; i < nSize; ++i)
                  pArrayffli“ = rand();
            break;
         case 0 :
#ifdef    STATIC_LINK
              printf("The array will now be freed.\n");
             free(pArray);
             _CRT_term();
```

```
#endif
            break;
        default   :
             printf("ulFlag = %lu\n", ulFlag);
            return  0UL;
    }

    /* A non-zero value must be returned to indicate success. */

    return  1UL;
}
#ifndef   STATIC_LINK
static void cleanup(void)
{
    printf("The array will now be freed.\n");
    free(pArray);
    return  ;
}
#endif
```

Building Dynamic Link Libraries in the *Programming Guide*

"_CRT_term — Terminate DLL Runtime Environment" on page 107
"_DLL_InitTerm — Initialize and Terminate DLL Environment" on page 144
"_rmem_init — Initialize Memory Functions for Subsystem DLL" on page 470
"_rmem_term — Terminate Memory Functions for Subsystem DLL" on page 476

_CRT_term — Terminate DLL Runtime Environment

Format

```
void _CRT_term(void);
/* no header file - defined in runtime startup code */
```

Description

Language Level: Extension

_CRT_term terminates the VisualAge for C++ runtime environment.

By default, all DLLs call the VisualAge for C++ _DLL_InitTerm function, which in turn calls _CRT_term for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than runtime initialization and termination), you must call _CRT_term from your _DLL_InitTerm function. If _CRT_init has been called, there must be a matching _CRT_term to insure library termination.

If your DLL contains C++ code, you must also call __ctordtorTerm **before** you call _CRT_term to ensure that static constructors and destructors are terminated correctly.

The prototype for __ctordtorTerm is void_ctordtorTerm(int flag). Value 0 for flag means that destructors are called for process termination. Value 1 means that only destructors for thread specific objects in the current thread should be called.

Once you have called _CRT_term, you cannot call any other library functions.

_CRT_term flushes the file buffers, but does not perform a complete library shutdown until the last user of the library calls _CRT_term. This is done by incrementing a *use counter* in _CRT_init, and decrementing it in _CRT_term. When the count reaches zero, the complete library shutdown occurs (files and semaphores are closed and the heap is destroyed). This prevents premature library shutdown because the DLL termination order is always predictable.

Failing to provide a matching call to _CRT_term could, under some circumstances, cause a DLL loaded by DosLoadModule or _loadmod, to fail to release its resources when it is freed.

Return Value

There is no return value for _CRT_term.

_CRT_term

This example shows the _DLL_InitTerm function from the VisualAge for C++ sample program for building DLLs, which calls _CRT_term to terminate the library environment.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define   CCHMAXPATH   255
/* _CRT_init is the C run-time environment initialization function.          */
/* It will return 0 to indicate success and -1 to indicate failure.          */

int _CRT_init(void);
#ifdef     STATIC_LINK

/* _CRT_term is the C run-time environment termination function.             */
/* It only needs to be called when the C run-time functions are statically   */
/* linked.                                                                   */
void _CRT_term(void);
#else


static void    cleanup(void);
#endif
size_t nSize;
int *pArray;

/* _DLL_InitTerm is the function that gets called by the operating system    */
/* loader when it loads and frees this DLL for each process that accesses    */
/* this DLL.    However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.    The system      */
/* linkage convention MUST be used because the operating system loader is    */
/* calling this function.                                                    */

unsigned long _System _DLL_InitTerm(unsigned long hModule, unsigned long
                                    ulFlag, long * dummy)
{
    size_t  i;
    int    rc;
    char  namebuffflCCHMAXPATH";
```

```
    /* If ulFlag is one then the DLL is being loaded so initialization should*/
    /* be performed.    If ulFlag is zero then the DLL is being freed so          */
    /* termination should be performed.                                          */

    switch (ulFlag) {
        case 1 :

            /*************************************************************************/
             /* The C run-time environment initialization function must be          */
             /* called before any calls to C run-time functions that are not       */
             /*  inlined.                                                           */
            /*************************************************************************/

             if (_CRT_init() == -1)
                 return  0UL;
#ifndef   STATIC_LINK

            /*************************************************************************/
             /* Cleanup routine is registered by atexit function to clean up       */
             /* if the runtime is dynamically linked.                              */
            /*************************************************************************/

                 if  (atexit(cleanup))
                      printf("atexit for cleanup failed\n");
#endif
             if ((rc = GetModuleFileName((void *)hModule, namebuf, CCHMAXPATH)) == 0)
                  printf("GetModuleFileName returned %lu\n", GetLastError());
            else
                  printf("The name of this DLL is %s\n", namebuf);
            srand(17);
             nSize = (rand()%128)+32;
             printf("The array size for this process is %u\n", nSize);
             if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
                  printf("Could not allocate space for unsorted array.\n");
                 return  0UL;
            }
             for (i = 0; i < nSize; ++i)
                  pArrayffli“ = rand();
            break;
         case 0 :
#ifdef    STATIC_LINK
             printf("The array will now be freed.\n");
            free(pArray);
            _CRT_term();
```

```
#endif
            break;
        default   :
             printf("ulFlag = %lu\n", ulFlag);
            return  0UL;
    }

    /* A non-zero value must be returned to indicate success. */

    return  1UL;
}
#ifndef   STATIC_LINK
static void cleanup(void)
{
    printf("The array will now be freed.\n");
    free(pArray);
    return  ;
}
#endif
```

Building Dynamic Link Libraries in the *Programming Guide*

"_CRT_init — Initialize DLL Runtime Environment" on page 103
"_DLL_InitTerm — Initialize and Terminate DLL Environment" on page 144
"_rmem_init — Initialize Memory Functions for Subsystem DLL" on page 470
"_rmem_term — Terminate Memory Functions for Subsystem DLL" on page 476

csid — Determine Character Set ID for Multibyte Character

Format

```
#include <stdlib.h>
int csid(const char *c);
```

Description

Language Level: Extension

csid queries the locale and determines the character-set identifier for the specified character *c*.

Return Value

csid returns the character-set identifier, or −1 if the character is not valid.

This example checks the character-set ID for a character.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *string = "A";
   int     rc;

    rc = csid(string);
    printf("character '%s' is in character set id %i\n", string, rc);
   return  0;

   /*************************************************************************
        The output should be similar to :

        character 'A' is in character set id 0
   *************************************************************************/
}
```

"wcsid — Determine Character Set ID for Wide Character" on page 721

"<stdlib.h>" on page 775

_cscanf — Read Data from Keyboard

Format

```
#include <conio.h>
int _cscanf(char *format-string, argument-list);
```

Description

Language Level: Extension

_cscanf reads data directly from the keyboard to the locations given by *argument-list*, if any are specified. The _cscanf function uses the _getche function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*.

The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the scanf function. See "scanf — Read Data" on page 486 for a description of the *format-string*.

Note: Although _cscanf normally echoes the input character, it does not do so if the last action was a call to _ungetch.

Return Value

_cscanf returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at the end of the file. A return value of 0 means that no fields were assigned.

This example uses _cscanf to read strings from the screen.

```
#include <conio.h>

int main(void)
{
   char  buffer[24];

    _cprintf("\nPlease enter a filename:\n");
   _cscanf("%23s",  buffer);
    _cprintf("\nThe file name you entered was %23s.", buffer);
   return  0;

   /*************************************************************************
        The output should be similar to :

        Please enter a filename:
                                    file.dat
        The filename you entered was                file.dat.
   *************************************************************************/
}
```

"fscanf — Read Formatted Data" on page 245
"_getch - _getche — Read Character from Keyboard" on page 272
"scanf — Read Data" on page 486

"sscanf — Read Data" on page 530
"_ungetch — Push Character Back to Keyboard" on page 683
"<conio.h>" on page 762

ctime — Convert Time to Character String

Format

```
#include <time.h>
char *ctime(const time_t *time);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ctime converts the time value pointed to by *time* to local time in the form of a character string. A time value is usually obtained by a call to the time function.

The string result produced by ctime contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

For example:

```
Mon Jul 16 02:03:55 1987\n\0
```

ctime uses a 24-hour clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have a constant width. Dates with only one digit are preceded with a blank space. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Return Value

ctime returns a pointer to the character string result. There is no error return value. A call to ctime is equivalent to:

```
asctime(localtime(&anytime))
```

Note: The asctime, ctime, and other time functions may use a common, statically allocated buffer for holding the return string. Each call to one of these functions may destroy the result of the previous call.

This example polls the system clock using ctime. It then prints a message giving the current date and time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t  ltime;

   time(&ltime);
    printf("The time is %s", ctime(&ltime));
   return  0;

   /*****************************************************************************
        The output should be similar to :

        The time is Thu Dec 15 18:10:23 1994
   *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"strftime — Convert to Formatted Time" on page 557
"time — Determine Current Time" on page 625
"<time.h>" on page 779

_cwait — Wait for Child Process

Format

#include <process.h>
int _cwait(int *_stat_loc_, int _process_id_, int _action_code_);

Description

Language Level: Extension

_cwait delays the completion of a parent process until the child process specified by _process_id_ ends.

The _process_id_ is the value returned by the _spawn function that started the child process. If the specified child process ends before _cwait is called, _cwait returns to the calling process immediately with a value of −1. If the value of _process_id_ is 0, the parent process waits until all of its child processes end.

If the variable pointed to by _stat_loc_ is NULL, _cwait does not use it. If it is not NULL, _cwait places information about the return status and the return code of the child process in the location pointed to by _stat_loc_.

If the child process ended normally with a call to the Win32 ExitProcess API, the lowest-order byte of the variable pointed to by _stat_loc_ is 0. The next highest-order byte contains the lowest-order byte of the argument passed to ExitProcess by the child process. The value of this byte depends on how the child process caused the system to call ExitProcess. If the child called exit, _exit, or return from main, or used a ExitProcess coded into the program, the byte contains the lowest-order byte of the argument the child passed to exit, _exit, or return. The value of the byte is undefined if the child caused a ExitProcess call simply by reaching the end of main.

If the child process ended abnormally (without a call to ExitProcess), the lowest-order byte of the variable pointed to by _stat_loc_ contains the return code from the Win32 GetExitCodeProcess API, and the next higher-order byte is 0.

The _action_code_ specifies when the parent process is to start running again. Values for _action_code_ include:

Action Code	Meaning
WAIT_CHILD	The parent process waits until the specified child process ends.
WAIT_GRANDCHILD	The parent process waits until the child process and all of the child processes of that process end.

The action code values are defined in <process.h>.

An alternative to this function is the WaitForSingleObject API call.

Return Value

At the normal end of the child process, _cwait returns the process identifier of the child to the parent process. If a child process ends abnormally, _cwait returns -1 to the parent process and sets errno to EINTR. In the case of an error, _cwait returns immediately with a value of -1 and sets errno to one of the following values:

Value	Meaning
EINVAL	Incorrect action code.
ECHILD	No child process exists, or the process identifier is incorrect.

This example creates a new process called child.exe. The parent calls _cwait and waits for the child to end. The parent then displays the child's return information in hexadecimal.

```
#include <stdio.h>
#include <process.h>
#include <errno.h>

int stat_child;

int main(void)
{
   int  i,result;

    /* spawn a child and 'cwait' for it to finish                                    */

    if ((result = _spawnl(P_NOWAIT, "child", "child", "1", NULL)) != -1) {
        if ((i = _cwait(&stat_child, result, WAIT_CHILD)) != result)
            printf("Error ...expected pid from child");
        else  {
            if (0 == errno) {
                printf("Child process ended successfully and ...\n");
                printf("program returned to the Parent process.\n");
            }
            else
                printf("Child process had an error\n");
        }
   }
```

```
    else
        printf("Error ...could not spawn a child process\n");
    return  0;

    /*************************************************************************

        If the source code for child.exe is:

       #include  <stdio.h>

        int main(void) {

            puts("This line was written by child.exe");
           return  0;
        }

        The output should be similar to :

        This line was written by child.exe
        Child process ended successfully and ...
        program returned to the Parent process.
    *************************************************************************/
}
```

"exit — End Program" on page 179
"_exit — End Process" on page 180
"_spawnl - _spawnvpe — Start and Run Child Processes" on page 519
return
"<process.h>" on page 772

__cxchg — Exchange Character Value with Memory

Format

```
#include <builtin.h>
signed char _Builtin __cxchg(volatile signed char *lockptr,
                             signed char value);
```

Description

Language Level: Extension

__cxchg puts the specified *value* in the memory location pointed to by *lockptr*, and returns the value that was previously in that location.

Use this function to implement fast-RAM semaphores to serialize access to a critical resource (so that only one thread can use it at a time).

You can also use Win32 semaphores to serialize resource access, but they are slower. Typically you would create both a fast-RAM semaphore and an Win32 semaphore for the resource. For more details about Win32 semaphores and how to use them, see the *Win32 Programmer's Reference*.

To implement a fast-RAM semaphore, allocate a volatile memory location (for __cxchg, it must be a signed char), and statically initialize it to 0 to indicate that the resource is free (not being used). To give a thread access to the resource, call __cxchg from the thread to exchange a nonzero value with that memory location. If __cxchg returns 0, the thread can access the resource; it has also set the memory location so that other threads can tell the resource is in use. When your thread no longer needs the resource, call __cxchg again to reset the memory location to 0.

If __cxchg returns a nonzero value, another thread is already using the resource, and the calling thread must wait for access. You could then use the Windows semaphore to inform your waiting thread when the resource is unlocked by the thread currently using it.

It is important that you set the memory to a nonzero value when you are using the resource. You can use the same nonzero value for all threads, or a unique value for each.

Note: __cxchg is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of __cxchg.
You cannot parenthesize a call to __cxchg. (Parentheses specify a call to the function's backing code, and __cxchg has none.)

Return Value __cxchg returns the previous value stored in the memory location pointed to by *lockptr*.

This example creates five separate threads, each associated with a State. At most two threads can have a State of Eating at the same time. When a thread calls PickUpChopSticks, that function checks the states of the preceding threads to make sure they are not already Eating. If the call to __cxchg is successful, the thread has locked the Lock semaphore and can change its State to Eating. It then calls __cxchg a second time to unlock the semaphore, and returns.

If __cxchg cannot lock the semaphore, another thread has it locked. The current thread then suspends itself briefly with Sleep and tries again. The semaphore is used to ensure that two threads cannot simultaneously change their State to Eating, which would cause a deadlock. (Note that although the semaphore solves the problem of deadlock, it is not an optimal solution; some threads may never get the semaphore or the State of Eating.)

Note: You must compile this example with the multithread (/Gm) option. The example runs in an infinite loop; press Ctrl-C to end it.

```
   #pragma strings(readonly)

#if   (1 == __TOS_OS2__)
       #define    INCL_DOS                                  /*  For OS/2  */
       #include <os2.h>
       #define SLEEP   DosSleep(1)
#else
      #include <windows.h>                                 /*  For Windows  */
      #define SLEEP   Sleep(1)
#endif

   #include <stdlib.h>
   #include <stdio.h>
   #include <builtin.h>

    typedef enum {
       Thinking,
       Eating,
       Hungry
   } States;
```

```
#define UNLOCKED 0
#define  LOCKED     1
#define NUM_PHILOSOPHERS 5

static volatile unsigned char Lock;
static States State[NUM_PHILOSOPHERS];
static const int NameMap[NUM_PHILOSOPHERS] = {0, 1, 2, 3, 4};
static const char * const Names[NUM_PHILOSOPHERS] = {"Plato",
                                                     "Socrates",
                                                     "Kant",
                                                     "Hegel",
                                                     "Nietsche"};

void PickUpChopSticks(int Ident);
void PutDownChopSticks(int Ident);
void Think(int Ident);
void Eat(int Ident);
void _Optlink StartPhilosopher( void *pIdent );

void _Optlink StartPhilosopher( void *pIdent )
{
     int Ident = *(int *)pIdent;
     while (1) {
          State[Ident] = Hungry;
          printf("%s hungry.\n", Names[Ident]);
          PickUpChopSticks(Ident);
          Eat(Ident);
          PutDownChopSticks(Ident);
          Think(Ident);
     }
}

int main(int argc, char *argv[], char *envp[])
{
    int                  i;
     unsigned long tid;

     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
          tid = _beginthread( StartPhilosopher, NULL, 8192, (void*)&NameMap[i]);
          if (tid == -1) {

               printf("Unable to start %s.\n", Names[i]);
          }
          else  {
               printf("%s started with Thread Id = %d.\n", Names[i], tid);
          }
     }
```

__cxchg

```
#if    (1 == __TOS_OS2__)
        DosWaitThread(&tid,  DCWW_WAIT);
#else
        WaitForSingleObject((HANDLE)tid,  INFINITE);
#endif
        return  0;
    }

    void PickUpChopSticks(int Ident)
    {
         while (1) {
              if (State[(Ident + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] != Eating &&
                    State[(Ident + 1) % NUM_PHILOSOPHERS] != Eating                    &&
                    !__cxchg(&Lock, LOCKED)) {
                       State[Ident] = Eating;
                      __cxchg(&Lock,  UNLOCKED);
                      return;
              }
              else  {
                   SLEEP;
              }
         }
    }

    void PutDownChopSticks(int Ident)
    {
         State[Ident] = Thinking;
    }

    void Eat(int Ident)
    {
         printf("%s eating.\n", Names[Ident]);
        SLEEP;
    }

    void Think(int Ident)
    {
         printf("%s thinking.\n", Names[Ident]);
        SLEEP;
    }
```

```
/*****************************************************************************
        The output should be similar to :

        Plato started with Thread Id = 2.
        Socrates started with Thread Id = 3.
        Kant started with Thread Id = 4.
        Hegel started with Thread Id = 5.
        Nietsche started with Thread Id = 6.
       Plato  hungry.
       Plato  eating.
       Socrates  hungry.
       Kant  hungry.
       Kant  eating.
       Hegel  hungry.
       Nietsche  hungry.
       Kant  thinking.
       Plato  thinking.
       Plato  hungry.
       Plato  eating.
       Kant  hungry.
       Kant  eating.
           :

*****************************************************************************/
```

"__lxchg — Exchange Integer Value with Memory" on page 369
"__sxchg — Exchange Integer Value with Memory" on page 615
"<builtin.h>" on page 761

_debug_calloc — Allocate and Initialize Memory

Format

```
#include <stdlib.h>      /* also in <malloc.h> */
void *_debug_calloc(size_t num, size_t size,
                                const char *file, size_t line);
```

Description

Language Level: Extension

_debug_calloc is the debug version of calloc. Like calloc, it allocates memory from the default heap for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0.

In addition, _debug_calloc makes an implicit call to _heap_check, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the _heap_check and _dump_allocated or _dump_allocated_delta functions.

To use _debug_calloc, you must compile with the debug memory (/Tm) compiler option. This option maps all calloc calls to _debug_calloc (you can also call _debug_calloc explicitly).

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by _debug_calloc, use _debug_realloc and _debug_free; you can also use realloc and free if you do not want debug information about the operation.

A heap-specific version of this function (_debug_ucalloc) is also available. _debug_calloc always allocates memory from the default heap.

Return Value

_debug_calloc returns a pointer to the reserved space. If not enough memory is available, or if *num* or *size* is 0, _debug_calloc returns NULL.

This example reserves storage of 100 bytes. It then attempts to write to storage that was not allocated. When _debug_calloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the calloc calls to _debug_calloc.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)calloc(1, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr1, 'a', 105);       /* overwrites storage that was not allocated       */
    ptr2 = (char*)calloc(2, 20);          /* this call to calloc invokes _heap_check          */
    puts("_debug_calloc did not detect that a memory block was overwritten.");
   return  0;

   /*****************************************************************************
        The output should be similar to:

        End of allocated object 0x00073890 was overwritten at 0x000738f4.
        The first eight bytes of the memory block (in hex) are: 6161616161616161.
        This memory block was (re)allocated at line number 9 in _debug_callo.c.
        Heap state was valid at line 9 of _debug_callo.c.
        Memory error detected at line 14 of _debug_callo.c.
   *****************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"<malloc.h>" on page 769
"<stdlib.h>" on page 775
"calloc — Reserve and Initialize Storage" on page 75
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"_debug_free — Release Memory" on page 126
"_debug_malloc — Allocate Memory" on page 130
"_debug_realloc — Reallocate Memory Block" on page 132
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heap_check — Validate Default Memory Heap" on page 291

_debug_free — Release Memory

Format

```
#include <stdlib.h>       /* also in <malloc.h> */
void _debug_free(void *ptr, const char *file,
                                    size_t  line);
```

Description

Language Level: Extension

_debug_free is the debug version of free. Like free, it frees the block of memory pointed to by *ptr*. _debug_free also sets each block of freed memory to 0xFB, so you can easily locate instances where your program uses the data in freed memory.

In addition, _debug_free makes an implicit call to the _heap_check, and stores the file name *file* and the line number *line* where the memory is freed. This information can be used later by the _heap_check and _dump_allocated or _dump_allocated_delta functions.

To use _debug_free, you must compile with the debug memory (/Tm) compiler option. This option maps all free calls to _debug_free (you can also call _debug_free explicitly).

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because _debug_free always checks what heap the memory was allocated from, you can use _debug_free to free memory blocks allocated by the regular, heap-specific, or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, _debug_free generates an error message and the program ends.

Return Value

There is no return value.

This example reserves two blocks, one of 10 bytes and the other of 20 bytes. It then frees the first block and attempts to overwrite the freed storage. When _debug_free is called a second time, _heap_check detects the error, prints out several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the free calls to _debug_free.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)malloc(10)) || NULL == (ptr2 = (char*)malloc(20))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    free(ptr1);
    memset(ptr1, 'a', 5);          /* overwrites storage that has been freed          */
    free(ptr2);                    /* this call to free invokes _heap_check           */
    puts("_debug_free did not detect that a freed memory block was overwritten.");
    return  0;

/*****************************************************************************
        The output should be similar to:

        Free heap was overwritten at 0x00073890.
        Heap state was valid at line 12 of _debug_free.c.
        Memory error detected at line 14 of _debug_free.c.
*****************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_malloc — Allocate Memory" on page 130
"_debug_realloc — Reallocate Memory Block" on page 132
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"free — Release Storage Blocks" on page 238
"_heap_check — Validate Default Memory Heap" on page 291
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_debug_heapmin — Release Unused Memory in the Default Heap

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
int _debug_heapmin(const char *file, size_t line);
```

Description

Language Level: Extension

_debug_heapmin is the debug version of _heapmin. Like _heapmin, It returns all unused memory from the default runtime heap to the operating system.

In addition, _debug_heapmin makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the memory is returned. This information can be used later by the _heap_check function.

To use _debug_heapmin, you must compile with the debug memory (/Tm) compiler option. This option maps all _heapmin calls to _debug_heapmin (you can also call _debug_heapmin explicitly).

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

A heap-specific version of this function (_debug_uheapmin) is also available. _debug_heapmin always operates on the default heap.

Return Value

If successful, _debug_heapmin returns 0; otherwise, it returns -1.

This example allocates 10000 bytes of storage, changes the storage size to 10 bytes, and then uses _debug_heapmin to return the unused memory to the operating system. The program then attempts to overwrite memory that was not allocated. When _debug_heapmin is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the _heapmin calls to _debug_heapmin.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *ptr;

    /* Allocate a large object from the system */
    if (NULL == (ptr = (char*)malloc(100000))) {
        puts("Could not allocate memory block.");
       exit(EXIT_FAILURE);
   }
    ptr = (char*)realloc(ptr, 10);
    _heapmin();                         /* No allocation problems to detect                    */

    *(ptr - 1) = 'a';                /* Overwrite memory that was not allocated          */
    _heapmin();                         /* This call to _heapmin invokes _heap_check       */

    puts("_debug_heapmin did not detect that a non-allocated memory block"
           "was  overwritten.");
   return  0;

   /*************************************************************************
        Possible output is:

        Header information of object 0x000738b0 was overwritten at 0x000738ac.
        The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
        This memory block was (re)allocated at line number 13 in _debug_heapm.c.
        Heap state was valid at line 14 of _debug_heapm.c.
        Memory error detected at line 17 of _debug_heapm.c.
   *************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_heapmin — Release Unused Memory in the Default Heap" on page 128
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heap_check — Validate Default Memory Heap" on page 291
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_debug_malloc — Allocate Memory

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void *_debug_malloc(size_t size,
                                const char *file, size_t line);
```

Description

Language Level: Extension

_debug_malloc is the debug version of malloc. Like malloc, it reserves a block of storage of *size* bytes from the default heap. _debug_malloc also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, _debug_malloc makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the storage is allocated. This information can later be used by the _heap_check and _dump_allocated or _dump_allocated_delta functions.

To use _debug_malloc, you must compile with the debug memory (/Tm) compiler option. This option maps all malloc calls to _debug_malloc (you can also call _debug_malloc explicitly).

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by _debug_malloc, use _debug_realloc and _debug_free; you can also use realloc and free if you do not want debug information about the operation.

A heap-specific version of this function (_debug_umalloc) is also available.

_debug_malloc always allocates memory from the default heap.

Return Value

_debug_malloc returns a pointer to the reserved space. If not enough memory is available or if *size* is 0, _debug_malloc returns NULL.

This example allocates 100 bytes of storage. It then attempts to write to storage that was not allocated. When _debug_malloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the malloc calls to _debug_malloc.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)malloc(100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    *(ptr1 - 1) = 'a';              /* overwrites storage that was not allocated       */
    ptr2 = (char*)malloc(10); /* this call to malloc invokes _heap_check           */
    puts("_debug_malloc did not detect that a memory block was overwritten.");
   return  0;

   /*************************************************************************
        Possible output is:

        Header information of object 0x00073890 was overwritten at 0x0007388c.
        The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
        This memory block was (re)allocated at line number 8 in _debug_mallo.c.
        Heap state was valid at line 8 of _debug_mallo.c.
        Memory error detected at line 13 of _debug_mallo.c.
   *************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_free — Release Memory" on page 126
"_debug_realloc — Reallocate Memory Block" on page 132
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heap_check — Validate Default Memory Heap" on page 291
"malloc — Reserve Storage Block" on page 376
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_debug_realloc — Reallocate Memory Block

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void *_debug_realloc(void *ptr, size_t size,
                                const char *file, size_t line);
```

Description

Language Level: Extension

_debug_realloc is the debug version of realloc. Like realloc, it reallocates the block of memory pointed to by *ptr* to a new *size*, specified in bytes. It also sets any new memory it allocates to 0xAA, so you can easily locate instances where your program tries to use the data in that memory without initializing it first.

In addition, _debug_realloc makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the storage is reallocated. This information can be used later by the _heap_check and _dump_allocated or _dump_allocated_delta functions.

If *ptr* is NULL, _debug_realloc behaves like _debug_malloc (or malloc) and allocates the block of memory.

To use _debug_realloc, you must compile with the debug memory (/Tm) compiler option. This option maps all realloc calls to _debug_realloc (you can also call _debug_realloc explicitly).

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because _debug_realloc always checks what heap the memory was allocated from, you can use _debug_realloc to reallocate memory blocks allocated by the regular, or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, _debug_realloc generates an error message and the program ends.

Return Value

_debug_realloc returns a pointer to the reallocated memory block. The *ptr* argument to _debug_realloc is not the same as the return value; _debug_realloc always changes the memory location to help you locate references to the memory that were not freed before the memory was reallocated.

If *size* is 0, _debug_realloc returns NULL. If not enough memory is available to expand the block to the given *size*, the original block is unchanged and NULL is returned.

This example uses _debug_realloc to allocate 100 bytes of storage. It then attempts to write to storage that was not allocated. When _debug_realloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the realloc calls to _debug_realloc.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char  *ptr;

    if (NULL == (ptr = (char*)realloc(NULL, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'a', 105);          /* overwrites storage that was not allocated        */
    ptr = (char*)realloc(ptr, 200);              /*    realloc invokes _heap_check          */
    puts("_debug_realloc did not detect that a memory block was overwritten." );
   return  0;

   /*************************************************************************
        The output should be similar to:

        End of allocated object 0x00073890 was overwritten at 0x000738f4.
        The first eight bytes of the memory block (in hex) are: 6161616161616161.
        This memory block was (re)allocated at line number 8 in _debug_reall.c.
        Heap state was valid at line 8 of _debug_reall.c.
        Memory error detected at line 13 of _debug_reall.c.
   *************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_free — Release Memory" on page 126
"_debug_malloc — Allocate Memory" on page 130
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heap_check — Validate Default Memory Heap" on page 291
"realloc — Change Reserved Storage Block Size" on page 452
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_debug_ucalloc — Reserve and Initialize Memory from User Heap

Format

```
#include <umalloc.h>
void *_debug_ucalloc(Heap_t heap, size_t num, size_t size,
                                const char *file, size_t line);
```

Description

Language Level: Extension

_debug_ucalloc is the debug version of _ucalloc. Like _ucalloc, it allocates memory from the *heap* you specify for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0.

In addition, _debug_ucalloc makes an implicit call to _uheap_check, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the _uheap_check and _udump_allocated or _udump_allocated_delta functions.

To use _debug_ucalloc, you must compile with the debug memory (/Tm) compiler option. the /Tm compiler option. This option maps all _ucalloc calls to _debug_ucalloc (you can also call _debug_ucalloc explicitly).

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

_debug_ucalloc works just like _debug_calloc except that you specify the heap to use; _debug_calloc always allocates from the default heap.

If the *heap* does not have enough memory for the request, _debug_ucalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _debug_ucalloc, use the non-heap-specific _debug_realloc and _debug_free. These functions always check what heap the memory was allocated from.

Return Value

_debug_ucalloc returns a pointer to the reserved space. If *size* or *num* was specified as zero, or if your *getmore_fn* cannot provide enough memory, _debug_ucalloc returns NULL. Passing _debug_ucalloc a heap that is not valid results in undefined behavior.

This example creates a user heap and allocates memory from it with _debug_ucalloc. It then attempts to write to memory that was not allocated. When _debug_free is called, _uheap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the /Tm option to map the _ucalloc calls to _debug_ucalloc and free to _debug_free.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
       exit(EXIT_FAILURE);
   }
    memset(ptr, 'x', 105);     /* Overwrites storage that was not allocated */
   free(ptr);
   return  0;

   /*************************************************************************
        The output should be similar to :

        End of allocated object 0x00073890 was overwritten at 0x000738f4.
        The first eight bytes of the memory block (in hex) are: 7878787878787878.
        This memory block was (re)allocated at line number 14 in _debug_ucallo.c.
        Heap state was valid at line 14 of _debug_ucallo.c.
        Memory error detected at line 19 of _debug_ucallo.c.
   *************************************************************************/
}
```

"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"calloc — Reserve and Initialize Storage" on page 75
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_free — Release Memory" on page 126
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"_ucalloc — Reserve and Initialize Memory from User Heap" on page 641
"_ucreate — Create a Memory Heap" on page 646
"_udump_allocated — Get Information about Allocated Memory in Heap" on page 656
"_uheap_check — Validate User Memory Heap" on page 662
"<umalloc.h>" on page 779

_debug_uheapmin — Release Unused Memory in User Heap

Format

#include <umalloc.h>
int _debug_uheapmin(Heap_t *heap*, const char **file*, size_t *line*);

Description

Language Level: Extension

_debug_uheapmin is the debug version of _uheapmin. Like _uheapmin, it returns all unused memory blocks from the specified *heap* to the operating system.

To return the memory, _debug_uheapmin calls the *release_fn* you supplied when you created the heap with _ucreate. If you do not supply a *release_fn*, _debug_uheapmin has no effect and returns 0.

In addition, _debug_uheapmin makes an implicit call to _uheap_check to validate the heap.

_debug_uheapmin works just like _debug_heapmin except that you specify the heap to use; _debug_heapmin always uses the default heap.

To use _debug_uheapmin, you must compile with the debug memory (/Tm) compiler option. This option maps all _uheapmin calls to _debug_uheapmin (you can also call _debug_uheapmin explicitly).

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value

If successful, _debug_uheapmin returns 0. A nonzero return value indicates failure. If the heap specified is not valid, _debug_uheapmin generates an error message with the file name and line number where the call to _debug_uheapmin was made.

This example creates a heap and allocates memory from it, then uses _debug_heapmin to release the memory.

Note: You must compile this example with the /Tm option to map the _uheapmin calls to _debug_uheapmin.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
    Heap_t    myheap;
    char      *ptr;

     /* Use default heap as user heap */
     myheap = _udefault(NULL);

     /* Allocate a large object */
     if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
          puts("Cannot allocate memory from user heap.\n");
          exit(EXIT_FAILURE);
     }
     memset(ptr, 'x', 60000);
    free(ptr);

     /* _debug_uheapmin will attempt to return the freed object to the system */
     if (0 != _uheapmin(myheap)) {
          puts("_debug_uheapmin returns failed.\n");
          exit(EXIT_FAILURE);
     }
    return  0;
}
```

"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_ucreate — Create a Memory Heap" on page 646
"_udump_allocated — Get Information about Allocated Memory in Heap" on page 656
"_uheap_check — Validate User Memory Heap" on page 662
"_uheapmin — Release Unused Memory in User Heap" on page 667
"<umalloc.h>" on page 779

_debug_umalloc — Reserve Memory Blocks from User Heap

Format

```
#include <umalloc.h>
void *_debug_umalloc(Heap_t heap, size_t size,
                                  const char *file, size_t line);
```

Description

Language Level: Extension

_debug_umalloc is the debug version of _umalloc. Like _umalloc, it reserves storage space from the *heap* you specify for a block of *size* bytes. _debug_umalloc also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, _debug_umalloc makes an implicit call to _uheap_check, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the _uheap_check and _udump_allocated or _udump_allocated_delta functions. _debug_umalloc also sets all the memory it allocates to 0xAA; this can help you debug problems where your program uses the data in the memory without initializing it.

_debug_umalloc works just like _debug_malloc except that you specify the heap to use; _debug_malloc always allocates from the default heap.

If the *heap* does not have enough memory for the request, _debug_umalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To use _debug_umalloc, you must compile with the debug memory (/Tm) compiler option. This option maps all _umalloc calls to _debug_umalloc (you can also call _debug_umalloc explicitly).

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated with _debug_umalloc, use the non-heap-specific _debug_realloc and _debug_free. These functions always check what heap the memory was allocated from.

Return Value

_debug_umalloc returns a pointer to the reserved space. If *size* was specified as zero, or the *getmore_fn* cannot provide enough memory, _debug_umalloc returns NULL. Passing _debug_umalloc a heap that is not valid results in undefined behavior.

This example creates a heap myheap and uses _debug_umalloc to allocate 100 bytes from it. It then attempts to overwrite storage that was not allocated. The call to

_debug_free invokes _uheap_check, which detects the error, generates messages, and ends the program.

Note: You must compile this example with the /Tm option to map _umalloc to _debug_umalloc and free to _debug_free.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
        puts("Cannot allocate memory from user heap.\n");
       exit(EXIT_FAILURE);
   }
    memset(ptr, 'x', 105);     /* Overwrites storage that was not allocated */
   free(ptr);
   return  0;

   /*************************************************************************
        The output should be similar to :

        End of allocated object 0x00073890 was overwritten at 0x000738f4.
        The first eight bytes of the memory block (in hex) are: 7878787878787878.
        This memory block was (re)allocated at line number 14 in _debug_umallo.c.
        Heap state was valid at line 14 of _debug_umallo.c.
        Memory error detected at line 19 of _debug_umallo.c.
   *************************************************************************/
}
```

"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_free — Release Memory" on page 126
"_debug_malloc — Allocate Memory" on page 130
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"malloc — Reserve Storage Block" on page 376
"_ucreate — Create a Memory Heap" on page 646
"_udump_allocated — Get Information about Allocated Memory in Heap" on page 656
"_umalloc — Reserve Memory Blocks from User Heap" on page 677
"_uheap_check — Validate User Memory Heap" on page 662
"<umalloc.h>" on page 779

difftime — Compute Time Difference

Format

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

difftime computes the difference in seconds between *time2* and *time1*.

Return Value

The difftime function returns the elapsed time in seconds from *time1* to *time2* as a double precision number. Type time_t is defined in <time.h>.

This example shows a timing application using difftime. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10000.

```
#include <time.h>
#include <stdio.h>

#define   RUNS              1000
#define   SIZE             10000

int mark[SIZE];

int main(void)
{
    time_t  start,finish;
    int  i,loop,n,num;

    time(&start);

     /*    This loop finds the prime numbers between 2 and SIZE                  */

     for (loop = 0; loop < RUNS; ++loop) {
          for (n = 0; n < SIZE; ++n)
               mark[n] = 0;

          /*    This loops marks all the composite numbers with -1               */
```

```
        for (num = 0, n = 2; n < SIZE; ++n)
            if (!mark[n]) {
                for (i = 2*n; i < SIZE; i += n)
                    mark[i] = -1;
                ++num;
            }
    }
    time(&finish);
     printf("Program takes an average of %f seconds to find %d primes.\n",
        difftime(finish, start)/RUNS, num);
    return  0;

    /*****************************************************************************
        The output should be similar to :

        Program takes an average of 0.106000 seconds to find 1229 primes.
    *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"strftime — Convert to Formatted Time" on page 557
"time — Determine Current Time" on page 625
"<time.h>" on page 779

_disable — Disable Interrupts

Format

```
#include <builtin.h>
void _disable( void );
```

Description

Language Level: Extension

_disable disables interrupts by executing the CLI machine instruction. It disables interrupts until the instruction after a call to _enable has been executed.

Note: _disable is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _disable.
You cannot parenthesize a call to _disable. (Parentheses specify a call to the function's backing code, and _disable has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

This function has no return value.

In this example, _disable disables interrupts by executing a CLI instruction.

```
#include <builtin.h>

int main(void)
{
    /* ------------------------------------------------------ */
    /* The expected assembler instruction looks like this :     */
    /*            CLI                                                    */
    /*  ------------------------------------------------------*/
    _disable();
    return  0;
}
```

"_enable — Enable Interrupts" on page 168
"_interrupt — Call Interrupt Procedure" on page 314
"<builtin.h>" on page 761

div — Calculate Quotient and Remainder

Format

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

div calculates the quotient and remainder of the division of *numerator* by *denominator*.

Return Value

div returns a structure of type div_t, containing both the quotient int quot and the remainder int rem. If the return value cannot be represented, its value is undefined. If *denominator* is 0, an exception will be raised.

This example uses div to calculate the quotients and remainders for a set of two dividends and two divisors.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int num[2] =    { 45,-45 };
    int den[2] =    { 7,-7 };
    div_t ans;                      /* div_t is a struct type containing two ints:
                                            'quot' stores quotient; 'rem' stores remainder */
   short  i,j;

    printf("Results of division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++) {
            ans = div(num[i], den[j]);
            printf("Dividend: %6ld    Divisor: %6ld", num[i], den[j]);
            printf("    Quotient: %6ld    Remainder: %6ld\n", ans.quot, ans.rem);
        }
   return  0;

   /*************************************************************************
        The output should be:

        Results of division:
        Dividend:   45   Divisor:     7   Quotient:     6   Remainder:     3
        Dividend:   45   Divisor:    -7   Quotient:    -6   Remainder:     3
        Dividend:  -45   Divisor:     7   Quotient:    -6   Remainder:    -3
        Dividend:  -45   Divisor:    -7   Quotient:     6   Remainder:    -3
   *************************************************************************/
}
```

"ldiv — Perform Long Division" on page 341
"<stdlib.h>" on page 775

_DLL_InitTerm — Initialize and Terminate DLL Environment

Format

```
unsigned long __stdcall
_DLL_InitTerm(HINSTANCE modhandle,
                                    DWORD flag, LPVOID dummy);
/* no header file - defined in runtime startup code */
```

Description

Language Level: Extension

_DLL_InitTerm is the initialization and termination entry point for a DLL. When each new process or thread gains access to the DLL, _DLL_InitTerm initializes the necessary environment for the DLL, including storage, semaphores, and variables. When each process or thread frees its access to the DLL, _DLL_InitTerm terminates the DLL environment created for that process or thread.

The default _DLL_InitTerm function supplied by VisualAge for C++ performs the actions required to initialize and terminate the runtime environment, or for subsystem DLLs, to initialize and terminate memory functions. It is called automatically when you link to the DLL. If your DLL requires initialization or termination actions in addition to the actions performed in the default function, you will need to create your own _DLL_InitTerm function. The *flag* parameter has four values defined in windows.h. If the value of the *flag* parameter is:

DLL_PROCESS_ATTACH The DLL environment is initialized

DLL_PROCESS_DETACH The DLL environment is ended

DLL_THREAD_ATTACH A new thread that can access the DLL has been created in the process.

DLL_THREAD_DETACH A thread has terminated

The *modhandle* parameter is the module handle assigned by the operating system for this DLL. You can use the module handle as a parameter to various Win32 API calls. For example, you can call GetModuleFileName with the module handle to return the fully-qualified path name of the DLL, which tells you where the DLL was loaded from.

Because it is called by the operating system loader, you must compile your _DLL_InitTerm function with __stdcall linkage.

For more information on creating your own _DLL_InitTerm function, see the chapter Building Dynamic Link Libraries in the *Programming Guide*.

Note: A _DLL_InitTerm function for a subsystem DLL has the same prototype, but the content of the function is different because there is no runtime environment to

initialize or terminate. For more information on building subsystem DLLs, see the section Building a Subsystem DLL in the *Programming Guide*.

Return Value The return code is significant only when the value of the *flag* parameter is DLL_PROCESS_ATTACH. The return code from _DLL_InitTerm tells the loader if the initialization was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of 0 indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

This function is also called by the operating system in thread creation and termination. When the value of the *flag* parameter is DLL_THREAD_ATTACH, a new THREAD has been created in the process. When the value of the *flag* parameter is DLL_THREAD_DETACH, a thread has terminated.

This example shows the _DLL_InitTerm function from the VisualAge for C++ sample program for building DLLs.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define   CCHMAXPATH   255
/* _CRT_init is the C run-time environment initialization function.          */
/* It will return 0 to indicate success and -1 to indicate failure.          */

int _CRT_init(void);
#ifdef     STATIC_LINK

/* _CRT_term is the C run-time environment termination function.             */
/* It only needs to be called when the C run-time functions are statically   */
/* linked.                                                                   */
void _CRT_term(void);
#else


static void    cleanup(void);
#endif
size_t nSize;
int *pArray;
```

_DLL_InitTerm

```
/* _DLL_InitTerm is the function that gets called by the operating system      */
/* loader when it loads and frees this DLL for each process that accesses     */
/* this DLL.    However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.    The system       */
/* linkage convention MUST be used because the operating system loader is     */
/* calling this function.                                                     */

unsigned long _System _DLL_InitTerm(unsigned long hModule, unsigned long
                                                    ulFlag, long * dummy)
{
    size_t  i;
    int    rc;
    char  namebuffflCCHMAXPATH“;

     /* If ulFlag is one then the DLL is being loaded so initialization should*/
     /* be performed.    If ulFlag is zero then the DLL is being freed so       */
     /* termination should be performed.                                       */

     switch (ulFlag) {
          case 1 :

              /************************************************************************/
               /* The C run-time environment initialization function must be          */
               /* called before any calls to C run-time functions that are not       */
              /*  inlined.                                                            */
              /************************************************************************/

               if (_CRT_init() == -1)
                   return  0UL;
#ifndef   STATIC_LINK

              /************************************************************************/
               /* Cleanup routine is registered by atexit function to clean up       */
               /* if the runtime is dynamically linked.                              */
              /************************************************************************/

                   if  (atexit(cleanup))
                        printf("atexit for cleanup failed\n");
```

```
#endif
             if ((rc = GetModuleFileName((void *)hModule, namebuf, CCHMAXPATH)) == 0)
                  printf("GetModuleFileName returned %lu\n", GetLastError());
             else
                  printf("The name of this DLL is %s\n", namebuf);
             srand(17);
              nSize = (rand()%128)+32;
              printf("The array size for this process is %u\n", nSize);
              if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
                   printf("Could not allocate space for unsorted array.\n");
                  return  0UL;
             }
              for (i = 0; i < nSize; ++i)
                   pArrayffli“ = rand();
             break;
        case 0 :
#ifdef    STATIC_LINK
              printf("The array will now be freed.\n");
             free(pArray);
             _CRT_term();
#endif
             break;
        default    :
             printf("ulFlag = %lu\n", ulFlag);
             return  0UL;
    }

    /* A non-zero value must be returned to indicate success. */

   return  1UL;
}
#ifndef   STATIC_LINK
static void cleanup(void)
{
    printf("The array will now be freed.\n");
   free(pArray);
   return  ;
}
#endif
```

The following example shows the _DLL_InitTerm function from the VisualAge for C++ sample for building subsystem DLLs.

_DLL_InitTerm

```
#pragma strings( readonly )
/* This example provides 5 external entrypoints of which 2 are exported for      */
/* use by client processes.    The entrypoints are:                                  */
/*                                                                                   */
/* _DLL_InitTerm - Initialization/Termination function for the DLL that is       */
/*                          invoked by the loader.    When the /Ge- compile option is       */
/*                          used the linker is told that this is the function to          */
/*                          execute when the DLL is first loaded and last freed for       */
/*                          each  process.                                                */
/*                                                                                   */
/* DLLREGISTER    - Called by _DLL_InitTerm for each process that loads the        */
/*                          DLL.                                                          */
/*                                                                                   */
/* DLLINCREMENT - Accepts a count from its client and adds this value to          */
/*                          the process and system totals.                                */
/*                                                                                   */
/* DLLSTATS         - Dumps process and system totals on behalf of its client.       */
/*                                                                                   */
/* DLLDEREGISTER- Called by _DLL_InitTerm for each process that frees the        */
/*                          DLL.                                                          */
/*                                                                                   */
/*****************************************************************************/

#include <windows.h>
#include <stdio.h>
#include "sample05.h"

unsigned long _System _DLL_InitTerm( unsigned long hModule, \
                                        unsigned long ulFlag, LPVOID dummy);

static unsigned long DLLREGISTER( void );
static unsigned long DLLDEREGISTER( void );

#define SHARED_SEMAPHORE_NAME "DLL.LCK"

/* The following data will be per-process data.    It will not be shared among    */
/* different processes.                                                              */

static HANDLE hmtxSharedSem;       /* Shared semaphore top                           */
static ULONG ulProcessTotal;     /* Total of increments for a process              */
static DWORD pid;                  /* Process identifier                             */

/* This is the global data segment that is shared by every process.              */

#pragma data_seg( GLOBAL_SEG )

static ULONG ulProcessCount;                  /* total number of processes          */
static ULONG ulGrandTotal;                    /* Grand total of increments          */
```

```
/* _DLL_InitTerm() - called by the loader for DLL initialization/termination    */
/* This function must return a non-zero value if successful and a zero value    */
/* if unsuccessful.                                                              */

unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag,
                                          LPVOID  dummy)
    {
    ULONG  rc;

     /* If ulFlag is one    then initialization is required:                      */
     /*       If the shared memory pointer is NULL then the DLL is being loaded    */
     /*       for the first time so acquire the named shared storage for the       */
     /*       process control structures.    A linked list of process control     */
     /*       structures will be maintained.    Each time a new process loads this */
     /*       DLL, a new process control structure is created and it is inserted   */
     /*       at the end of the list by calling DLLREGISTER.                       */
     /*                                                                            */
     /* If ulFlag is zero then termination is required:                            */
     /*       Call DLLDEREGISTER which will remove the process control structure   */
     /*       and free the shared memory block from its virtual address space.     */

     switch( ulFlag )
         {
    case  1:
              if ( !ulProcessCount )
                  {
                  _rmem_init();
                   /* Create the shared mutex semaphore.                          */

                   if (( hmtxSharedSem = CreateMutex(NULL,
                                                     TRUE,
                                                       SHARED_SEMAPHORE_NAME)) == NULL)
                       {
                        printf( "CreateMutex    rc = %lu\n", GetLastError() );
                       return  0;
                       }
                  }

              /* Register the current process.                                   */

              if ( DLLREGISTER( ) )
                  return  0;

             break;

         case  0:
              /* De-register the current process.                                */
```

```
            if ( DLLDEREGISTER( ) )
                return  0;

            _rmem_term();

            break;

        default:
            return  0;
        }

    /* Indicate success.    Non-zero means success!!!                                    */

    return  1;
    }

/* DLLREGISTER - Registers the current process so that it can use this          */
/*                       subsystem.    Called by _DLL_InitTerm when the DLL is first    */
/*                       loaded for the current process.                                */

static unsigned long DLLREGISTER( void )
    {
    ULONG  rc;

    /* Get the process id          */

      pid = GetCurrentProcessId();

    /* Open the shared mutex semaphore for this process.                              */

    if ( ( hmtxSharedSem = OpenMutex(SYNCHRONIZE,
                                     TRUE,
                                      SHARED_SEMAPHORE_NAME ) ) == NULL)
        {
         printf( "OpenMutex rc = %lu\n", rc = GetLastError() );
        return  rc;
        }

    /* Acquire the shared mutex semaphore.                                            */

    if ( WaitForSingleObject( hmtxSharedSem,
                                    INFINITE )    == 0xFFFFFFFF )
        {
         printf( "WaitFor SingleObject rc = %lu\n", rc = GetLastError() );
         CloseHandle( hmtxSharedSem );
        return  rc;
        }
```

```
    /* Increment the count of processes registered.                    */

    ++ulProcessCount;

    /* Initialize the per-process data.                                */

    ulProcessTotal = 0;

    /* Tell the user that the current process has been registered.     */

    printf( "\nProcess %lu has been registered.\n\n", pid );

    /* Release the shared mutex semaphore.                             */

    if ( ReleaseMutex( hmtxSharedSem ) == FALSE )
        {
         printf( "ReleaseMutex rc = %lu\n", rc = GetLastError() );
        return  rc;
        }

    return  0;
    }

/* DLLINCREMENT - Increments the process and global totals on behalf of the  */
/*                calling  program.                                          */

int DLLINCREMENT( int incount )
    {
     ULONG              rc;                  /* return code from DOS APIs    */

    /* Acquire the shared mutex semaphore.                             */

    if ( WaitForSingleObject( hmtxSharedSem,
                              INFINITE )    == 0xFFFFFFFF )
        {
         printf( "WaitForSingleObject rc = %lu\n", rc = GetLastError());
        return  rc;
        }

    /* Increment the counts the process and grand totals.              */

    ulProcessTotal += incount;
    ulGrandTotal += incount;

    /* Tell the user that the increment was successful.                */

    printf( "\nThe increment for process %lu was successful.\n\n", pid );

    /* Release the shared mutex semaphore.                             */
```

```
    if ( ReleaseMutex( hmtxSharedSem )    == FALSE )
        {
        printf( "ReleaseMutex rc = %lu\n", rc = GetLastError() );
        return  rc;
        }

    return  0;
    }

/* DLLSTATS    - Prints process and grand totals.                          */

int DLLSTATS( void )
    {
    ULONG                   rc;

    /* Acquire the shared mutex semaphore.                                  */

    if ( WaitForSingleObject( hmtxSharedSem,
                                          INFINITE )    == 0xFFFFFFFF )
        {
        printf( "WaitForSingleObject rc = %lu\n", rc = GetLastError());
        return  rc;
        }

    /* Print out per-process and global information.                        */

    printf( "\nCurrent process identifier          = %lu\n", pid );
    printf( "Current process total                 = %lu\n", ulProcessTotal );
    printf( "Number of processes registered = %lu\n", ulProcessCount );
    printf( "Grand Total                              = %lu\n\n", ulGrandTotal );

    /* Release the shared mutex semaphore.                                  */

    if ( ReleaseMutex( hmtxSharedSem ) == FALSE )
        {
        printf( "ReleaseMutex rc = %lu\n", rc = GetLastError() );
        return  rc;
        }

    return  0;
    }

/* DLLDEREGISTER - Deregisters the current process from this subsystem.    */
/*                         Called by _DLL_InitTerm when the DLL is freed for the */
/*                         last time by the current process.              */

static unsigned long DLLDEREGISTER( void )
    {
    ULONG  rc;

    /* Acquire the shared mutex semaphore.                                  */
```

```
 if ( ( rc = WaitForSingleObject( hmtxSharedSem,
                                          INFINITE ) ) == 0xFFFFFFFF )
     {
      printf( "WaitForSingleObject rc = %lu\n", GetLastError());
     return  rc;
     }

 /* Decrement the count of processes registered.                              */

--ulProcessCount;

 /* Tell the user that the current process has been deregistered.          */

 printf( "\nProcess %lu has been deregistered.\n\n", pid );

 /* Release the shared mutex semaphore.                                       */

 if ( ReleaseMutex( hmtxSharedSem ) == FALSE )
     {
      printf( "ReleaseMutex rc = %lu\n", rc= GetLastError() );
     return  rc;
     }

 /* Close the shared mutex semaphore for this process.                        */

 if ( CloseHandle( hmtxSharedSem )    == FALSE )
     {
      printf( "CloseHandle      rc = %lu\n", rc = GetLastError() );
     return  rc;
     }

return  0;
}
```

Building a Dynamic Link Library in the *Programming Guide*
Building a Subsystem DLL in the *Programming Guide*
"_CRT_init — Initialize DLL Runtime Environment" on page 103
"_CRT_term — Terminate DLL Runtime Environment" on page 107
"_rmem_init — Initialize Memory Functions for Subsystem DLL" on page 470
"_rmem_term — Terminate Memory Functions for Subsystem DLL" on page 476

_dump_allocated — Get Information about Allocated Memory

Format

```
#include <stdlib.h>     /* also in <malloc.h> */
void _dump_allocated(int nbytes);
```

Description

Language Level: Extension

_dump_allocated prints information to **stderr** about each memory block that is currently allocated and was allocated using the debug memory management functions (_debug_calloc, _debug_malloc, and so on).

Use *nbytes* to specify how many bytes of each memory block are to be printed. If *nbytes* is a:

Negative value	Prints all bytes of each block.
0	Prints no bytes.
Positive value	Prints the specified number of bytes or the entire block, whichever is smaller.

Call _dump_allocated at points in your code where you want a report of the currently allocated memory. For example, a good place to call _dump_allocated is a point where most of the memory is already freed and you want to find a memory leak, such as at the end of a program.

_dump_allocated prints the information to **stderr**, but you can change the destination with the _set_crt_msg_handle function. You can also use _dump_allocated_delta to display information only about the memory that was allocated since the previous call to _dump_allocated or _dump_allocated_delta.

To use _dump_allocated and the debug memory management functions, you must compile with the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

A heap-specific version of this function (_udump_allocated) is also available. _dump_allocated always prints information about memory allocated from the default heap.

Return Value There is no return value.

This example allocates three memory blocks, and then calls _dump_allocated to dump information and the first 8 bytes for each memory block.

Note: You must compile this example with the /Tm option to map the memory management functions to their debug versions.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define   INIT_STR          "It\0works"
#define   INIT_STR_LEN   (sizeof(INIT_STR)  -  1)

int main(void) {
    char  *pBlock1;
    char  *pBlock2;
    char  *pBlock3;

     if (NULL == (pBlock1 = (char*)malloc(35))) { /* allocate first memory block*/
          printf("Could not allocate first memory block.\n");
         return    EXIT_FAILURE;
    }

     memcpy(pBlock1, INIT_STR, INIT_STR_LEN); /* initialize first memory block    */
     if (NULL == (pBlock2 = (char*)calloc(2, 120))) { /* allocate second
                                                                                  memory  block    */
          printf("Could not allocate second memory block.\n");
         return    EXIT_FAILURE;
    }

     memcpy(pBlock2, INIT_STR, INIT_STR_LEN); /* initialize second memory block */
     if (NULL == (pBlock3 = (char*)realloc(NULL, 2235))) {    /* allocate third
                                                                          memory  block         */
          printf("Could not allocate third memory block.\n");
         return    EXIT_FAILURE;
    }

     memcpy(pBlock3, INIT_STR, INIT_STR_LEN); /* initialize third memory block    */
     if (NULL == (pBlock3 = (char*)realloc(pBlock3, 300))) {/* reallocate third */
                                              /* memory block to different size            */

          printf("Could not reallocate third memory block.\n");
         return    EXIT_FAILURE;
    }

     _dump_allocated(8);       /* show first eight bytes of each memory block          */
    return  0;
```

```
/*****************************************************************************
    The output should be similar to:

    -------------------------------------------------------------------------
                        START OF DUMP OF ALLOCATED MEMORY BLOCKS
    -------------------------------------------------------------------------
     Address: 0x00074310         Size: 0x0000012C (300)
     This memory block was (re)allocated at line number 32 in _dump_alloc.c.
    Memory  contents:   49740077  6F726B73                    [It.works        ]
    -------------------------------------------------------------------------
     Address: 0x000738F0         Size: 0x000000F0 (240)
     This memory block was (re)allocated at line number 19 in _dump_alloc.c.
    Memory  contents:   49740077  6F726B73                    [It.works        ]
    -------------------------------------------------------------------------
     Address: 0x00073890         Size: 0x00000023 (35)
     This memory block was (re)allocated at line number 13 in _dump_alloc.c.
    Memory  contents:   49740077  6F726B73                    [It.works        ]
    -------------------------------------------------------------------------
                        END OF DUMP OF ALLOCATED MEMORY BLOCKS
    -------------------------------------------------------------------------
  *****************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_free — Release Memory" on page 126
"_debug_malloc — Allocate Memory" on page 130
"_debug_realloc — Reallocate Memory Block" on page 132
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heap_check — Validate Default Memory Heap" on page 291
"_udump_allocated — Get Information about Allocated Memory in Heap" on page 656
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_dump_allocated_delta — Get Information about Allocated Memory

Format

```
#include <stdlib.h>        /* also in <malloc.h> */
void _dump_allocated_delta(int nbytes);
```

Description

Language Level: Extension

_dump_allocated_delta prints information to **stderr** about each memory block allocated by a debug memory management function (_debug_malloc and so on) since the last call to _dump_allocated_delta or _dump_allocated.

_dump_allocated_delta and _dump_allocated print the same type of information to **stderr**, but _dump_allocated displays information about all blocks that have been allocated since the start of your program.

Use *nbytes* to specify how many bytes of each memory block are to be printed. If *nbytes* is a:

Negative value	Prints all bytes of each block.
0	Prints no bytes.
Positive value	Prints the specified number of bytes or the entire block, whichever is smaller.

_dump_allocated_delta prints the information to **stderr**, but you can change the destination with the _set_crt_msg_handle function.

A heap-specific version of this function, _udump_allocated_delta, is also available. _dump_allocated_delta always displays information about the default heap.

To use _dump_allocated_delta and the debug versions of the memory management functions, specify the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) are mapped to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value

There is no return value.

This example allocates some memory and calls _dump_allocated to print information about the allocated blocks. It then allocates more memory, and calls _dump_allocated_delta again to print information about the allocations since the previous call.

Note: You must compile this example with the /Tm option to map the memory management functions to their debug versions.

_dump_allocated_delta

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)malloc(10))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr1, 'a', 5);
    _dump_allocated(10);

    if (NULL == (ptr2 = (char*)malloc(20))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr2, 'b', 5);
    _dump_allocated_delta(10);

    free(ptr1);
    free(ptr2);
    return  0;

    /*************************************************************************
        The output should be similar to :

        -------------------------------------------------------------------------
                            START OF DUMP OF ALLOCATED MEMORY BLOCKS
        -------------------------------------------------------------------------
        Address: 0x00073890         Size: 0x0000000A (10)
        This memory block was (re)allocated at line number 9 in _dump_alloc_d.c.
        Memory contents:   61616161 61AAAAAA AAAA                   [aaaaaêêêêê         ]
        -------------------------------------------------------------------------
                            END OF DUMP OF ALLOCATED MEMORY BLOCKS
        -------------------------------------------------------------------------
        -------------------------------------------------------------------------
                            START OF DUMP OF ALLOCATED MEMORY BLOCKS
        -------------------------------------------------------------------------
        Address: 0x000738D0         Size: 0x00000014 (20)
        This memory block was (re)allocated at line number 16 in _dump_alloc_d.c.
        Memory contents:   62626262 62AAAAAA AAAA                   [bbbbbêêêêê         ]
        -------------------------------------------------------------------------
                            END OF DUMP OF ALLOCATED MEMORY BLOCKS
        -------------------------------------------------------------------------
    *************************************************************************/
}
```

"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_malloc — Allocate Memory" on page 130
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_debug_realloc — Reallocate Memory Block" on page 132
"_set_crt_msg_handle — Change Runtime Message Output Destination" on page 495

"_udump_allocated_delta — Get Information about Allocated Memory in Heap" on page 659
"<stdlib.h>" on page 775
"<malloc.h>" on page 769

dup — Associate Second Handle with Open File

Format

```
#include <io.h>
int dup(int handle);
```

Description

Language Level: XPG4, Extension

dup associates a second file handle with a currently open file. You can carry out operations on the file using either file handle because all handles associated with a given file use the same file pointer. Creation of a new handle does not affect the type of access allowed for the file.

For example, given:

handle2 = dup(*handle1*)

handle2 will have the same file access mode (text or binary) as *handle1*. In addition, if *handle1* was originally opened with the O_APPEND flag (described in "open — Open File" on page 418), *handle2* will also have that attribute.

Warning: Both handles share a single file pointer. If you reposition a file using *handle1*, the position in the file returned by *handle2* will also change.

If you duplicate a file handle for an open stream, the resulting file handle has the same restrictions as the original file handle.

Note: In earlier releases of VisualAge C++, dup began with an underscore (_dup). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _dup to dup for you.

Return Value

dup returns the next available file handle for the given file. It returns –1 if an error occurs and sets errno to one of the following values:

Value	Meaning
EBADF	The file handle is not valid.
EMFILE	No more file handles are available.
EOS2ERR	The call to the operating system was not successful.

This example makes a second file handle, fh3, refer to the same file as the file handle fh1 using dup. The file handle fh2 is then associated with the file edopen.da2, and finally fh2 is forced to associate with edopen.da1 dup2.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>

int main(void)
{
    int  fh1,fh2,fh3;

     if (-1 == (fh1 = open("edopen.da1", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
         ))  {
          perror("Unable to open edopen.da1");
         return  EXIT_FAILURE;
    }
     if (-1 == (fh3 = dup(fh1))) {          /* fh3 refers to the sample file as fh1    */
          perror("Unable to dup");
         close(fh1);
         return  EXIT_FAILURE;
    }
    else
          printf("Successfully performed dup handle.\n");
     if (-1 == (fh2 = open("edopen.da2", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
         ))  {
          perror("Unable to open edopen.da2");
         close(fh1);
         close(fh3);
         return  EXIT_FAILURE;
    }
     if (-1 == dup2(fh1, fh2)) {    /* Force fh2 to the refer to the same file        */
                                                  /* as fh1.                                                    */
          perror("Unable to dup2");
    }
    else
          printf("Successfully performed dup2 handle.\n");
    close(fh1);
    close(fh2);
    close(fh3);
    return  0;

    /*************************************************************************
          The output should be:

            Successfully performed dup handle.
            Successfully performed dup2 handle.
    *************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"creat — Create New File" on page 100
"dup2 — Associate Second Handle with Open File" on page 163

dup

"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"<io.h>" on page 764

dup2 — Associate Second Handle with Open File

Format

```
#include <io.h>
int dup2(int handle1, int handle2);
```

Description

Language Level: XPG4, Extension

dup2 makes *handle2* refer to the currently open file associated with *handle1*. You can carry out operations on the file using either file handle because all handles associated with a given file use the same file pointer.

handle2 will point to the same file as *handle1*, but will retain the file access mode (text or binary) that it had before duplication. In addition, if *handle2* was originally opened with the O_APPEND flag, it will also retain that attribute. If *handle2* is associated with an open file at the time of the call, that file is closed.

Warning: Both handles share a single file position. If you reposition a file using *handle1*, the position in the file returned by *handle2* will also change.

If you duplicate a file handle for an open stream (using fileno), the resulting file handle has the same restrictions as the original file handle.

Note: In earlier releases of VisualAge C++, dup2 began with an underscore (_dup2). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _dup2 to dup2 for you.

Return Value

dup2 returns 0 to indicate success. It returns −1 if an error occurs and sets errno to one of the following values:

Value	Meaning
EBADF	The file handle is not valid.
EMFILE	No more file handles are available.
EOS2ERR	The call to the operating system was not successful.

dup2

This example makes a second file handle, fh3, refer to the same file as the file handle fh1 using dup. The file handle fh2 is then associated with the file edopen.da2, and finally fh2 is forced to associate with edopen.da1 by the dup2 function.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>

int main(void)
{
    int  fh1,fh2,fh3;

     if (-1 == (fh1 = open("edopen.da1", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
         ))  {
          perror("Unable to open edopen.da1");
         return  EXIT_FAILURE;
    }
     if (-1 == (fh3 = dup(fh1))) {          /* fh3 refers to the sample file as fh1    */
          perror("Unable to dup");
         close(fh1);
         return  EXIT_FAILURE;
    }
    else
          printf("Successfully performed dup handle.\n");
     if (-1 == (fh2 = open("edopen.da2", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
         ))  {
          perror("Unable to open edopen.da2");
         close(fh1);
         close(fh3);
         return  EXIT_FAILURE;
    }
     if (-1 == dup2(fh1, fh2)) {    /* Force fh2 to the refer to the same file        */
                                                  /* as fh1.                                                    */
          perror("Unable to dup2");
    }
    else
          printf("Successfully performed dup2 handle.\n");
    close(fh1);
    close(fh2);
    close(fh3);
    return  0;

    /*************************************************************************
          The output should be:

            Successfully performed dup handle.
            Successfully performed dup2 handle.
    *************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"creat — Create New File" on page 100
"dup — Associate Second Handle with Open File" on page 160

“open — Open File” on page 418
“_sopen — Open Shared File” on page 516
“<io.h>” on page 764

_ecvt — Convert Floating-Point to Character

Format

#include <stdlib.h>
char *_ecvt(double *value*, int *ndigits*, int **decptr*, int **signptr*);

Description

Language Level: Extension

_ecvt converts the floating-point number *value* to a character string. _ecvt stores *ndigits* digits of *value* as a string and adds a null character (\0). If the number of digits in *value* exceeds *ndigits*, the low-order digit is rounded. If there are fewer than *ndigits* digits, the string is padded with zeros. Only digits are stored in the string.

You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr*. *decptr* points to an integer value that gives the position of the decimal point with respect to the beginning of the string. A 0 or a negative integer value indicates that the decimal point lies to the left of the first digit.

signptr points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. If it is not 0, the number is negative.

_ecvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively. For more information on NaN and infinity values, see "Infinity and NaN Support" on page 27.

Warning: For each thread, the _ecvt, _fcvt and _gcvt functions use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Return Value

_ecvt returns a pointer to the string of digits. Because of the limited precision of the double type, no more than 16 decimal digits are significant in any conversion. If it cannot allocate memory to perform the conversion. _ecvt returns NULL and sets **errno** to ENOMEM.

This example reads in two floating-point numbers, calculates their product, and prints out only the billions digit of its character representation. At most, 16 decimal digits of significance can be expected. The output assumes the user enters the numbers 1000000 and 3000.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
    float  x,y;
    double  z;
    int  w,b,decimal,sign;
    char  *buffer;

     printf("Enter two floating-point numbers:\n");
     if (2 != scanf("%e %e", &x, &y)) {
         printf("input  error...\n");
         return  EXIT_FAILURE;
    }
     z = x *y;
     printf("Their product is %g\n", z);
     w = log10(fabs(z))+1.;
     buffer = _ecvt(z, w, &decimal, &sign);
     b = decimal-10;
     if (b < 0)
          printf("Their product does not exceed one billion.\n");
    else
          if (b > 15)
               printf("The billions digit of their product is insignificant.\n");
          else
               printf("The billions digit of their product is %c.\n", buffer[b]);
    return  0;

    /*****************************************************************************
          For the following input:

          1000000  3000

          The output should be:

          Enter two floating-point numbers:
          1000000  3000
          Their product is 3e+09
          The billions digit of their product is 3.
    *****************************************************************************/
}
```

"_fcvt — Convert Floating-Point to String" on page 192
"_gcvt — Convert Floating-Point to String" on page 268
"Infinity and NaN Support" on page 27
"<stdlib.h>" on page 775

_enable — Enable Interrupts

Format

```
#include <builtin.h>    /* also defined in <stdlib.h> */
void _enable( void );
```

Description

Language Level: Extension

_enable enables interrupts by generating the STI machine instruction. Interrupts are enabled after the instruction following STI has been executed. If interrupts are disabled and you call _enable followed immediately by a call to _disable, interrupts remain disabled.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _enable.
You cannot parenthesize an _enable function call because parentheses specify a call to the backing code for the function in the library.

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception will be generated.

Return Value

There is no return value.

In this example, _enable enables interrupts by executing an STI instruction.

```
#include <builtin.h>

int main(void)
{
    /* ------------------------------------------------------- */
    /* The expected assembler instruction looks like this :    */
    /*             STI                                         */
    /* ------------------------------------------------------- */
    _enable();
    return  0;
}
```

"_disable — Disable Interrupts" on page 142
"_interrupt — Call Interrupt Procedure" on page 314
"<builtin.h>" on page 761

_endthread — Terminate Current Thread

Format

```
#include <stdlib.h>    /* also in <process.h> */
void _endthread(void);
```

Description

Language Level: Extension

_endthread ends a thread that you previously created with _beginthread. When the thread has finished, it automatically ends itself with an implicit call to _endthread. You can also call _endthread explicitly to end the thread before it has completed its function, for example, if some error condition occurs.

Note: When using the _beginthread and _endthread functions, you must specify the /Gm+ compiler option to use the multithread libraries.

Return Value

There is no return value.

In this example, the main program creates two threads, bonjour and au_revoir.

Note: You must compile this example with the /Gm+ option.

```
/*****************************************************************************
 Note:    To run this example, you must compile it using
 the /Gm+ compile-time option.
 *****************************************************************************          */

#if (1 == __TOS_OS2__)
      #define   INCL_DOS                                  /*   OS/2   */
     #include  <os2.h>
#else
      #include  <windows.h>                               /*   Windows   */
#endif

#include <stdio.h>
#include <stdlib.h>
#include <process.h>

void _Optlink bonjour(void *arg)
{
    int i = 0;

    while (i++ < 5)
        printf("Bonjour!\n");
    _endthread();                                  /* This thread ends itself explicitly */
    puts("thread should terminate before printing this");
}
```

_endthread

```
void _Optlink au_revoir(void *arg)
{
    int i = 0;

    while (i++ < 5)                              /* This thread makes an implicit     */
        printf("Au revoir!\n");                  /*    call to _endthread             */
}

int main(void)
{
    unsigned long tid1;
    unsigned long tid2;

    tid1 = _beginthread(bonjour, NULL, 8192, NULL);
    tid2 = _beginthread(au_revoir, NULL, 8192, NULL);
    if (-1 == tid1 || -1 == tid2) {
        printf("Unable to start threads.\n");
       return  EXIT_FAILURE;
    }
#if (1 == __TOS_OS2__)
    DosWaitThread(&tid2, DCWW_WAIT);             /*    wait until threads 1 and 2 */
    DosWaitThread(&tid1, DCWW_WAIT);             /*    have been completed             */
#else
    WaitForSingleObject((HANDLE)tid2, INFINITE);   /* wait until threads 1 and 2 */
    WaitForSingleObject((HANDLE)tid1, INFINITE);   /* have been completed             */
#endif

   return  0;

}
```

"_beginthread — Create New Thread" on page 66
/Gm+ option. in the *User's Guide*
"<stdlib.h>" on page 775
"<process.h>" on page 772

__eof — Determine End of File

Format

```
#include <io.h>
int __eof (int handle);
```

Description

Language Level: Extension

__eof determines whether the file pointer has reached the end-of-file for the file associated with *handle*. You cannot use __eof on a nonseekable file; it will fail.

Return Value

__eof returns the value 1 if the current position is the end of the file or 0 if it is not. A return value of −1 shows an error, and the system sets errno to the following values:

Value	Meaning
EBADF	File handle is not valid.
EOS2ERR	The call to the operating system was not successful.

This example creates the file sample.dat and then checks if the file pointer is at the end of that file using the __eof function.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int  fh,returnValue;

    fh = creat("sample.dat", S_IREAD|S_IWRITE);
    if (-1 == fh) {
        perror("Error creating sample.dat");
       return  EXIT_FAILURE;
   }
    if (-1 == (returnValue = __eof(fh))) {
        perror("eof function error");
       return  EXIT_FAILURE;
   }
    if (1 == returnValue)
        printf("File pointer is at end-of-file position.\n");
   else
        printf("File pointer is not at end-of-file position.\n");
   close(fh);
   return  0;

   /*********************************************************************
        The output should be:

        File pointer is at end-of-file position.
   *********************************************************************/
}
```

__eof

"_chsize — Alter Length of File" on page 85
"creat — Create New File" on page 100
"_filelength — Determine File Length" on page 211
"_sopen — Open Shared File" on page 516
"open — Open File" on page 418
"<io.h>" on page 764

erf – erfc — Calculate Error Functions

Format

```
#include <math.h>
double erf(double x);
double erfc(double x);
```

Description

Language Level: SAA, XPG4

erf calculates the error function of

$$2\pi^{-1/2}\int_0^x e^{-t^2}\,dt$$

erfc computes the value of 1.0 - erf(*x*). erfc is used in place of erf for large values of *x*.

Return Value

erf returns a double value that represents the error function. erfc returns a double value representing 1.0 - erf.

This example uses erf and erfc to compute the error function of two numbers.

```
#include <stdio.h>
#include <math.h>

double smallx, largex, value;

int main(void)
{
      smallx = 0.1;
      largex = 10.0;
      value = erf(smallx);              /* value = 0.112463 */
      printf("Error value for 0.1: %lf\n", value);
      value = erfc(largex);             /* value = 2.088488e-45 */
      printf("Error value for 10.0: %le\n", value);
     return  0;


/*********************************************************************
*
Output should be:

Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45


                                                                     *
*********************************************************************/
}
```

erf - erfc

"Bessel Functions — Solve Differential Equations" on page 69
"gamma — Gamma Function" on page 267
"<math.h>" on page 770

execl - _execvpe — Load and Run Child Process

Format

```
#include <process.h>
int execl(char *pathname, char *arg0, char *arg1,...,
                  char *argn, NULL);
int execle(char *pathname, char *arg0, char *arg1,...,
                  char *argn, NULL, char *envp[ ]);
int execlp(char *pathname, char *arg0, char *arg1,...,
                  char *argn, NULL);
int _execlpe(char *pathname, char *arg0, char *arg1,...,
                  char *argn, NULL, char *envp[ ]);
int execv(char *pathname, char *argv[ ]);
int execve(char *pathname, char *argv[ ],char *envp[ ]);
int execvp(char *pathname, char *argv[ ]);
int _execvpe(char *pathname, char *argv[ ], char *envp[ ]);
```

Description

Language Level: XPG4 (except _execlpe and _execvpe), Extension

The exec functions load and run new child processes. The parent process is ended after the child process has started. Sufficient storage must be available for loading and running the child process.

All of the exec functions are versions of the same routine; the letters at the end determine the specific variation:

Letter	Variation
p	Uses PATH environment variable to find the file to be run.
l	Passes a list of command line arguments separately.
v	Passes to the child process an array of pointers to command-line arguments.
e	Passes to the child process an array of pointers to environment strings.

Note: In earlier releases of VisualAge C++, all of the exec functions began with an underscore (_execl). Because they are defined by the X/Open standard, the underscore has been removed. _execlpe and _execvpe retain the initial underscore because they are not included in the X/Open standard. For compatibility, VisualAge for C++ will map the _exec functions to the correct exec function.

The *pathname* argument specifies the file to run as the child process. The *pathname* can specify a full path from the root, a partial path from the current working directory, or a file name. If *pathname* does not have a file name extension or does not end with a period, the exec functions will add the .EXE extension and search for the file. If *pathname* has an extension, the exec function uses only that extension. If *pathname* ends with a period, the exec functions search for *pathname* with no extension. The execlp, _execlpe, execvp, and _execvpe functions search for the *pathname* in the directories that the PATH environment variable specifies.

You pass arguments to the new process by giving one or more pointers to character strings as arguments in the exec call. These character strings form the argument list for the child process.

The compiler can pass the argument pointers as separate arguments (execl, execle, execlp, and _execlpe) or as an array of pointers (execv, execve, execvp, and _execvpe). You should pass at least one argument, either *arg0* or *argv*[0], to the child process. If you do not, an argument will be returned that points to the same file as the path name argument you specified. This argument may not be exactly identical to the path name argument you specified. A different value does not produce an error.

Use the execl, execle, execlp, and _execlpe functions for the cases where you know the number of arguments in advance. The *arg0* argument is usually a pointer to *pathname*. The *arg1* through *argn* arguments are pointers to the character strings forming the new argument list. There must be a NULL pointer following argn to mark the end of the argument list.

Use the execv, execve, execvp, and _execvpe functions when the number of arguments to the new process is variable. Pass pointers to the arguments of these functions as an array, argv[ ]. The argv[0] argument is usually a pointer to *pathname*. The argv[1] through argv[*n*] arguments are pointers to the character strings forming the new argument list. If argv[*n*] is the last parameter, then argv[*n+1*] must be NULL.

Files that are open when you make an exec call remain open in the new process. In the execl, execlp, execv, and execvp calls, the child process receives the environment of the parent. The execle, _execlpe, execve, and _execvpe functions let you change the environment for the child process by passing a list of environment settings through the envp argument. The **envp** argument is an array of character pointers, each element of which points to a string ending with a null character that defines an environment variable. Such a string usually has the following form:

```
NAME=value
```

where *NAME* is the name of an environment variable, and *value* is the string value to which the exec function sets that variable.

Note: Do not enclose the *value* in double quotation marks.

The final element of the **envp** array should be NULL. When **envp** itself is NULL, the child process receives the environment settings of the parent process.

The exec functions do not preserve signal settings in child processes created by calls to exec functions. Calls to exec functions reset the signal settings to the default in the child process.

Return Value

The exec functions do not normally return control to the calling process. They are equivalent to the corresponding _spawn functions with P_OVERLAY as the value of *modeflag*. If an error occurs, the exec functions return -1 and set **errno** to one of the following values:

Value	Meaning
EACCESS	The specified file has a locking or sharing violation.
EMFILE	There are too many open files. The system must open the specified file to tell whether it is an executable file.
ENOENT	The file or *pathname* was not found or was specified incorrectly.
ENOEXEC	The specified file cannot run or has an incorrect executable file format.
ENOMEM	One of the following conditions exists: Not enough storage is available to run the child process. Not enough storage is available for the argument or environment strings.

This example calls four of the eight exec routines. When invoked without arguments, the program first runs the code for case PARENT. It then calls execle() to load and run a copy of itself. The instructions for the child are blocked to run only if argv[0] and one parameter were passed (case CHILD). In its turn, the child runs its own child as a copy of the same program. This sequence is continued until four generations of child processes have run. Each of the processes prints a message identifying itself.

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

#define   PARENT          1
#define   CHILD           2

char *args[3];

int main(int argc, char **argv, char **envp) {
   switch(argc)  {
      case PARENT: {                          /* No argument: run a child */
         printf("Parent process began.\n");
         execle(argv[0],argv[0],"1",NULL,envp);
         abort();        /* Not executed because parent was overlaid. */
      }
```

```
        case CHILD: {                   /* One argument: run a child's child */
            printf("Child process %s began.\n", argv[1]);
            if ('1' == *argv[1]) {      /* generation one */
                execl(argv[0], argv[0], "2", NULL);
                abort();        /* Not executed because child was overlaid */
            }
            if('2' == *argv[1]) {                       /* generation two */
                args[0] = argv[0];
                args[1] = "3";
                args[2] = NULL;
               execv(argv[0],args);
                abort();        /* Not executed because child was overlaid */
            }
            if ('3' == *argv[1]) {                      /* generation three */
                args[0] = argv[0];
                args[1] = "4";
                args[2] = NULL;
                execve(argv[0], args, _environ);
                abort();        /* Not executed because child was overlaid */
            }
            if ('4' == *argv[1])                        /* generation four */
                printf("Child process %s", argv[1]);
        }
    }
    printf("  ended.\n");
    return  55;
    /* The output should be similar to:
        Parent process began.
        Child process 1 began.
        Child process 2 began.
        Child process 3 began.
        Child process 4 began.
        Child process 4 ended.                                              */
}
```

"abort — Stop a Program" on page 39
"_cwait — Wait for Child Process" on page 116
"exit — End Program" on page 179
"_exit — End Process" on page 180
"_spawnl - _spawnvpe — Start and Run Child Processes" on page 519
"system — Invoke the Command Processor" on page 613
"Arguments to main" in the *Language Reference*
"<process.h>" on page 772

exit — End Program

Format

```
#include <stdlib.h>    /* also in <process.h> */
void exit(int status);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

exit returns control to the host environment from the program. It first calls all functions registered with the atexit function, in reverse order; that is, the last one registered is the first one called. See "atexit — Record Program Termination Function" on page 54 for more information. It flushes all buffers and closes all open files before ending the program. All files opened with tmpfile are deleted.

The argument *status* can have a value from 0 to 255 inclusive or be one of the macros EXIT_SUCCESS or EXIT_FAILURE. A *status* value of EXIT_SUCCESS or 0 indicates a normal exit; otherwise, another status value is returned.

Return Value

exit returns both control and the value of *status* to the operating system.

This example ends the program after flushing buffers and closing any open files if it cannot open the file myfile.mjq.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
    if (NULL == (stream = fopen("myfile.mjq", "r"))) {
        perror("Could not open data file");
        exit(EXIT_FAILURE);
    }
    return 0;

    /*************************************************************************
        The output should be:

        Could not open data file: The file cannot be found.
    *************************************************************************/
}
```

"abort — Stop a Program" on page 39
"atexit — Record Program Termination Function" on page 54
"_onexit — Record Termination Function" on page 416
"_exit — End Process" on page 180
"signal — Handle Interrupt Signals" on page 510
"tmpfile — Create Temporary File" on page 626
"<stdlib.h>" on page 775

_exit — End Process

Format

```
#include <stdlib.h>    /* also in <process.h> */
void _exit(int status);
```

Description

Language Level: Extension

_exit ends the calling process without calling functions registered by _onexit or atexit. It also does not flush stream buffers or delete temporary files. You supply the *status* value as a parameter; the value 0 typically means a normal exit.

Return Value

Although _exit does not return a value, the value is available to the waiting parent process, if there is one, after the child process ends. If no parent process waits for the exiting process, the *status* value is lost. The *status* value is available through the operating system batch command IF ERRORLEVEL.

This example calls _exit to end the process. Because _exit does not flush the buffer first, the output from the second printf statement will not appear.

```
#include <stdio.h>
#include <stdlib.h>                          /* You can also use <process.h>          */

char buf[51];

int main(void)
{

    /* Make sure the standard output stream is line-buffered even if the          */
    /* output is redirected to a file.                                                */

    if (0 != setvbuf(stdout, buf, _IOLBF, 50))
        printf("The buffering was not set correctly.\n");
    printf("This will print out but ...\n");
    printf("this will not!");
   _exit(EXIT_FAILURE);
   return  0;

   /*****************************************************************************
        The output should be:

        This will print out but ...
   *****************************************************************************/
}
```

"abort — Stop a Program" on page 39
"atexit — Record Program Termination Function" on page 54
"execl - _execvpe — Load and Run Child Process" on page 175
"exit — End Program" on page 179
"_onexit — Record Termination Function" on page 416
"<process.h>" on page 772
"<stdlib.h>" on page 775

exp — Calculate Exponential Function

Format

```
#include <math.h>
double exp(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

exp calculates the exponential function of a floating-point argument *x* (e^x, where *e* equals 2.17128128...).

Return Value

If an overflow occurs, exp returns HUGE_VAL. If an underflow occurs, it returns 0. Both overflow and underflow set errno to ERANGE.

This example calculates *y* as the exponential function of *x*:

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y;

    x = 5.0;
    y = exp(x);
    printf("exp( %lf ) = %lf\n", x, y);
   return  0;

   /*************************************************************************
        The output should be:

        exp( 5.000000 ) = 148.413159
   *************************************************************************/
}
```

"log — Calculate Natural Logarithm" on page 358
"log10 — Calculate Base 10 Logarithm" on page 359
"<math.h>" on page 770

fabs — Calculate Floating-Point Absolute Value

Format

```
#include <math.h>
double fabs(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fabs calculates the absolute value of the floating-point argument *x*.

Return Value

fabs returns the absolute value. There is no error return value.

This example calculates *y* as the absolute value of *x*:

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y;

    x = -5.6798;
    y = fabs(x);
    printf("fabs( %lf ) = %lf\n", x, y);
   return  0;

   /*************************************************************************
        The output should be similar to :

        fabs( -5.679800 ) = 5.679800
   *************************************************************************/
}
```

"abs — Calculate Integer Absolute Value" on page 40
"_cabs — Calculate Absolute Value of Complex Number" on page 73
"labs — Calculate Absolute Value of Long Integer" on page 339
"<math.h>" on page 770

_facos — Calculate Arccosine

Format

```
#include <builtin.h>
double _facos(double x);
```

Description

Language Level: Extension

_facos calculates the arccosine of *x*. This function causes the compiler to emit the appropriate 80387 instructions for the calculation of the arccosine.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _facos.
You cannot parenthesize a _facos function call, because parentheses specify a call to the backing code for the function.

Return Value

_facos returns the arccosine of *x*.

This example prompts for a value for *x*. It prints an error message if *x* is greater than 1 or less than -1. Otherwise, it assigns the arccosine of *x* to *y*.

```
#include <builtin.h>
#include <stdio.h>

#define MAX    1.0
#define MIN -1.0

int main(void)
{
   double  x;

   printf("Enter  x:\n");
   scanf("%lf",  &x);

    /* Output error if not in range */
    if (MAX < x)
        printf("Error: %lf too large for acos.\n", x);
    else if (MIN > x)
                printf("Error: %lf too small for acos.\n", x);
            else
                printf("The arccosine of %lf is %lf.\n", x, _facos(x));
   return  0;

   /*****************************************************************************
        Assuming you enter: -1.0

        The output should be:

        The arccossine of -1.000000 is 3.141593.
   *****************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43

_facos

"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_fcos — Calculate Cosine" on page 189
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"<builtin.h>" on page 761

_fasin — Calculate Arcsine

Format

```
#include <builtin.h>
double _fasin(double x);
```

Description

Language Level: Extension

_fasin calculates the arcsine of *x*. This function causes the compiler to emit the appropriate 80387 instructions for the calculation of arcsine.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fasin.
You cannot parenthesize a _fasin function call, because parentheses specify a call to the backing code for the function.

Return Value

_fasin returns the arcsine of *x*.

This example prompts for a value of *x*. It prints an error message if *x* is greater than 1 or less than -1. Otherwise, it assigns the arcsine of *x* to *y*.

```
#include <builtin.h>
#include <stdio.h>

#define MAX    1.0
#define MIN -1.0

int main(void)
{
   double  x;

   printf("Enter  x:\n");
   scanf("%lf",  &x);

    /* Output error if not in range */
    if (MAX < x)
         printf("Error: %lf too large for asin.\n", x);
    else if (MIN > x)
                printf("Error: %lf too small for asin.\n", x);
           else
                printf("The arcsine of %lf is %lf.\n", x, _fasin(x));
   return  0;

   /*****************************************************************************
         Assuming you enter: 1.0

         The ouput should be:

         The arcsine of 1.000000 is 1.570796.
   *****************************************************************************/
}
```

_fasin

"asin — Calculate Arcsine" on page 49
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsin — Calculate Sine" on page 251
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"<builtin.h>" on page 761

fclose — Close Stream

Format

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fclose closes a stream pointed to by *stream*. This function flushes all buffers associated with the stream before closing it. When it closes the stream, the function releases any buffers that the system reserved. When a binary stream is closed, the last record in the file is padded with null characters (\0) to the end of the record.

Return Value

fclose returns 0 if it successfully closes the stream, or EOF if any errors were detected.

Note: Once you close a stream with fclose, you must open it again before you can use it.

This example opens a file fclose.dat for reading as a stream; then it closes this file.

```
#include <stdio.h>

int main(void)
{
   FILE  *stream;

    stream = fopen("fclose.dat", "r");
    if (0 != fclose(stream))                              /* Close the stream.*/
        perror("fclose  error");
   else
        printf("File closed successfully.\n");
   return  0;

   /**************************************************************************
        The output should be:

        File closed successfully.
   **************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"_fcloseall — Close All Open Streams" on page 188
"fflush — Write Buffer to File" on page 199
"fopen — Open Files" on page 218
"freopen — Redirect Open Files" on page 242
"<stdio.h>" on page 774

_fcloseall — Close All Open Streams

Format

```
#include <stdio.h>
int _fcloseall(void);
```

Description

Language Level: Extension

_fcloseall closes all open streams, except stdin, stdout, and stderr. It also closes and deletes any temporary files created by tmpfile.

_fcloseall flushes all buffers associated with the streams before closing them. When it closes streams, it releases the buffers that the system reserved, but does not release user-allocated buffers.

Return Value

_fcloseall returns the total number of streams closed, or EOF if an error occurs.

This example opens a file john for reading as a data stream, and then closes the file. It closes all other streams except stdin, stdout, and stderr.

```
#include <stdio.h>

#define OUTFILE "temp.out"

int main(void)
{
   FILE  *stream;
   int  numclosed;

    stream = fopen(OUTFILE, "w");
    numclosed = _fcloseall();
    printf("Number of files closed = %d\n", numclosed);
   remove(OUTFILE);
   return  0;

   /*************************************************************************
       The output should be:

       Number of files closed = 1
   *************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"fclose — Close Stream" on page 187
"fflush — Write Buffer to File" on page 199
"fopen — Open Files" on page 218
"freopen — Redirect Open Files" on page 242
"tmpfile — Create Temporary File" on page 626
"<stdio.h>" on page 774

_fcos — Calculate Cosine

Format

```
#include <builtin.h>
double _fcos( double x);
```

Description

Language Level: Extension

_fcos calculates the cosine of *x*, where *x* is expressed in radians. The value of *x* must be less than 2**63. This function causes the compiler to emit the appropriate 80387 instruction and return only the cosine.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fcos.
You cannot parenthesize a _fcos function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

_fcos returns the cosine expressed in radians.

This example calculates *y* to be the cosine of *x*.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  x;

    x = 7.2;
    printf("The cosine of %lf is %lf.\n", x, _fcos(x));
   return  0;

   /*****************************************************************************
        The output should be similar to :

        The cosine of 7.200000 is 0.608351.
   *****************************************************************************/
}
```

"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_facos — Calculate Arccosine" on page 183
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"<builtin.h>" on page 761

_fcossin — Calculate Cosine and Sine

Format

```
#include <builtin.h>
double _fcossin(double x, double *y);
```

Description

Language Level: Extension

_fcossin calculates the cosine of *x*, and stores the sine of *x* in **y*. This is faster than separately calculating the sine and cosine. Use _fcossin instead of _fsincos when you will be using the cosine first, and then the sine. This function causes the compiler to emit the FSINCOS 80387 instruction.

Because it is a built-in function and has no backing code in the library:

- You cannot take the address of _fcossin.
- You cannot parenthesize a _fcossin function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

_fcossin returns the cosine of *x*.

This example calculates the cosine of x and stores it in z, and stores the sine of x in *y.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;

   printf("Enter  x:\n");
   scanf("%lf",  &x);

    z = _fcossin(x, &y);
    printf("The cosine of %lf is %lf.\n", x, z);
    printf("The sine of %lf is %lf.\n", x, y);
   return  0;

   /*****************************************************************************
        Assuming you enter : 1.0

        The output should be :

        The cosine of 1.000000 is 0.540302.
        The sine of 1.000000 is 0.841471.
   *****************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43
"asin — Calculate Arcsine" on page 49
"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97

"_facos — Calculate Arccosine" on page 183
"_fasin — Calculate Arcsine" on page 185
"_fcos — Calculate Cosine" on page 189
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"_fsin — Calculate Sine" on page 251
"_fsincos — Calculate Sine and Cosine" on page 252
"<builtin.h>" on page 761

_fcvt — Convert Floating-Point to String

Format

#include <stdlib.h>
char *_fcvt(double *value*, int *ndec*, int **decptr*, int **signptr*);

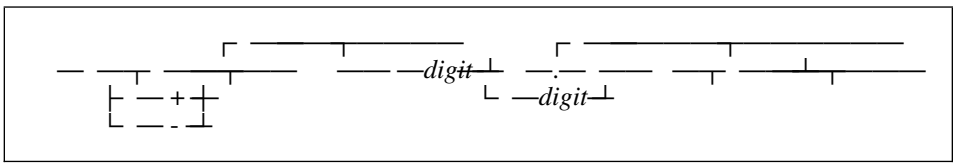
Description

Language Level: Extension

_fcvt converts the floating-point number *value* to a character string. _fcvt stores the digits of *value* as a string and adds a null character (\0). The *ndec* variable specifies the number of digits to be stored after the decimal point.

If the number of digits after the decimal point in *value* exceeds *ndec*, _fcvt rounds the correct digit according to the FORTRAN F format. If there are fewer than *ndec* digits of precision, _fcvt pads the string with zeros.

A FORTRAN F number has the following format:



_fcvt stores only digits in the string. You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr*. *decptr* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value shows that the decimal point lies to the left of the first digit.

signptr points to an integer showing the sign of *value*. _fcvt sets the integer to 0 if *value* is positive and to a nonzero number if *value* is negative.

_fcvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively. For more information on NaN and infinity values, see "Infinity and NaN Support" on page 27. **Warning:** For each thread, the _ecvt, _fcvt, and _gcvt functions use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Return Value

_fcvt returns a pointer to the string of digits. Because of the limited precision of the double type, no more than 16 decimal digits are significant in any conversion. If it cannot allocate memory to perform the conversion, _fcvt returns NULL and sets **errno** to ENOMEM.

This example reads in two floating-point numbers, computes their product, and prints out only the billions digit of its character representation. At most, 16 decimal digits of significance can be expected. The output given assumes the user enters the values 2000000 and 3000.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
    float x = 2000000;
    float y = 3000;
   double  z;
   int  w,b,decimal,sign;
   char  *buffer;

    z = x *y;
    printf("The product of %e and %e is %g.\n", x, y, z);
    w = log10(fabs(z))+1.;
    buffer = _fcvt(z, w, &decimal, &sign);
    b = decimal-10;
    if (b < 0)
        printf("Their product does not exceed one billion.\n");
    if (b > 15)
        printf("The billions digit of their product is insignificant.\n");
    if ((b > -1) && (b < 16))
        printf("The billions digit of their product is %c.\n", buffer[b]);
   return  0;

   /*************************************************************************
        The output should be:

        The product of 2.000000e+06 and 3.000000e+03 is 6e+09.
        The billions digit of their product is 6.
   *************************************************************************/
}
```

"_ecvt — Convert Floating-Point to Character" on page 166
"_gcvt — Convert Floating-Point to String" on page 268
"Infinity and NaN Support" on page 27
"<stdlib.h>" on page 775

fdopen — Associates Input Or Output With File

Format

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

Description

Language Level: XPG4, Extension

fdopen associates an input or output stream with the file identified by *handle*. The *type* variable is a character string specifying the type of access requested for the stream.

Mode	Description
r	Create a stream to read a text file. The file pointer is set to the beginning of the file.
w	Create a stream to write to a text file. The file pointer is set to the beginning of the file.
a	Create a stream to write, in append mode, at the end of the text file. The file pointer is set to the end of the file.
r+	Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
w+	Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
a+	Create a stream for reading or writing, in append mode, at the end of the text file. The file pointer is set to the end of the file.
rb	Create a stream to read a binary file. The file pointer is set to the beginning of the file.
wb	Create a stream to write to a binary file. The file pointer is set to the beginning of the file.
ab	Create a stream to write to a binary file in append mode. The file pointer is set to the end of the file.
r+b *or* rb+	Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
w+b *or* wb+	Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
a+b *or* ab+	Create a stream for reading and writing to a binary file in append mode. The file pointer is set to the end of the file.

Warning: Use the w, w+, wb, wb+, and w+b modes with care; they can destroy existing files.

The specified *type* must be compatible with the access mode you used to open the file. If the file was opened with the O_APPEND FLAG, the stream mode must be r, a, a+, rb, ab, a+b, or ab+.

When you open a file with a, a+, ab, a+b, or ab+ as the value of *type*, all write operations take place at the end of the file. Although you can reposition the file pointer using fseek or rewind, the file pointer always moves back to the end of the file before the system carries out any write operation. This action prevents you from writing over existing data.

When you specify any of the types containing +, you can read from and write to the file, and the file is open for update. However, when switching from reading to writing or from writing to reading, you must include an intervening fseek, fsetpos, or rewind operation. You can specify the current file position with fseek.

In accessing text files, carriage-return line-feed (CR-LF) combinations are translated into a single line feed (LF) on input; LF characters are translated to CR-LF combinations on output. Accesses to binary files suppress all of these translations. (See "Stream Processing" in the *Programming Guide* for the differences between text and binary streams.)

If fdopen returns NULL, use close to close the file. If fdopen is successful, you must use fclose to close the stream and file.

Note: In earlier releases of VisualAge C++, fdopen began with an underscore (_fdopen). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _fdopen to fdopen for you.

Return Value

fdopen returns a pointer to a file structure that can be used to access the open file. A NULL pointer return value indicates an error.

This example opens the file sample.dat and associates a stream with the file using fdopen. It then reads from the stream into the buffer.

fdopen

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
    long  length;
    int  fh;
    char  buffer[20];
    FILE  *fp;

     memset(buffer, '\0', 20);                                  /* Initialize buffer*/
    printf("\nCreating  sample.dat.\n");
     system("echo Sample Program > sample.dat");
     if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
          perror("Unable to open sample.dat");
         return  EXIT_FAILURE;
    }
     if (NULL == (fp = fdopen(fh, "r"))) {
         perror("fdopen  failed");
         close(fh);
         return  EXIT_FAILURE;
    }
     if (7 != fread(buffer, 1, 7, fp)) {
         perror("fread  failed");
         fclose(fp);
         return  EXIT_FAILURE;
    }
     printf("Successfully read from the stream the following:\n%s.\n", buffer);
    fclose(fp);
    return  0;

    /*****************************************************************************
          The output should be:

         Creating  sample.dat.
          Successfully read from the stream the following:
         Sample  .
    *****************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"creat — Create New File" on page 100
"fclose — Close Stream" on page 187
"fopen — Open Files" on page 218
"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"open — Open File" on page 418
"rewind — Adjust Current File Position" on page 465
"_sopen — Open Shared File" on page 516
"<stdio.h>" on page 774

feof — Test End-of-File Indicator

Format

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

feof indicates whether the end-of-file flag is set for the given *stream*. The end-of-file flag is set by several functions to indicate the end of the file. The end-of-file flag is cleared by calling rewind, fsetpos, fseek, or clearerr for this stream.

Return Value

feof returns a nonzero value if and only if the EOF flag is set; otherwise, it returns 0.

This example scans the input stream until it reads an end-of-file character.

```
#include <stdio.h>

int main(void)
{
   char  inp_char;
   FILE  *stream;

    stream = fopen("feof.dat", "r");

    /* scan an input stream until an end-of-file character is read                    */

    while (0 == feof(stream)) {
        fscanf(stream, "%c", &inp_char);
        printf("<x%x> ", inp_char);
   }
   fclose(stream);
   return  0;

   /*************************************************************************
        If feof.dat contains : abc defgh

        The output should be:

        <x61> <x62> <x63> <x20> <x64> <x65> <x66> <x67> <x68>
   *************************************************************************/
}
```

"clearerr — Reset Error Indicators" on page 87
"ferror — Test for Read/Write Errors" on page 198
"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"perror — Print Error Message" on page 427
"rewind — Adjust Current File Position" on page 465
"<stdio.h>" on page 774

ferror — Test for Read/Write Errors

Format

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ferror tests for an error in reading from or writing to the given *stream*. If an error occurs, the error indicator for the *stream* remains set until you close *stream*, call rewind, or call clearerr.

Return Value

The ferror function returns a nonzero value to indicate an error on the given *stream*. A return value of 0 means no error has occurred.

This example puts data out to a stream and then checks that a write error has not occurred.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE  *stream;
    char *string = "Important information";

    stream = fopen("ferror.dat", "w");
    fprintf(stream, "%s\n", string);
    if (ferror(stream)) {
        printf("write  error\n");
        clearerr(stream);
        return  EXIT_FAILURE;
   }
   else
         printf("Data written to a file successfully.\n");
   if  (fclose(stream))
        perror("fclose  error");
   return  0;

   /*************************************************************************
        The output should be:

        Data written to a file successfully.
   *************************************************************************/
}
```

"clearerr — Reset Error Indicators" on page 87
"feof — Test End-of-File Indicator" on page 197
"fopen — Open Files" on page 218
"perror — Print Error Message" on page 427
"strerror — Set Pointer to Runtime Error Message" on page 550
"_strerror — Set Pointer to System Error String" on page 551
"<stdio.h>" on page 774

fflush — Write Buffer to File

Format

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fflush causes the system to empty the buffer associated with the specified output *stream*, if possible. If the *stream* is open for input, fflush undoes the effect of any ungetc function. The *stream* remains open after the call.

If *stream* is NULL, the system flushes all open streams.

Note: The system automatically flushes buffers when you close the stream, or when a program ends normally without closing the stream.

Return Value

fflush returns the value 0 if it successfully flushes the buffer. It returns EOF if an error occurs.

This example flushes a stream buffer.

```
#include <stdio.h>

int main(void)
{
   FILE  *stream;

    stream = fopen("myfile.dat", "w");
    fprintf(stream, "Hello world");
   fflush(stream);
   fclose(stream);
   return  0;
}
```

"fclose — Close Stream" on page 187
"_flushall — Write Buffers to Files" on page 215
"setbuf — Control Buffering" on page 493
"ungetc — Push Character onto Input Stream" on page 681
"<stdio.h>" on page 774

fgetc — Read a Character

Format

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fgetc reads a single unsigned character from the input *stream* at the current position and increases the associated file pointer, if any, so that it points to the next character.

Note: fgetc is identical to getc but is always implemented as a function call; it is never replaced by a macro.

Return Value

fgetc returns the character read as an integer. An EOF return value indicates an error or an end-of-file condition. Use feof or ferror to determine whether the EOF value indicates an error or the end of the file.

This example gathers a line of input from a stream.

```
#include <stdio.h>

#define   MAX_LEN           80

int main(void)
{
   FILE  *stream;
   char  buffer[MAX_LEN+1];
   int  i,ch;

    stream = fopen("myfile.dat", "r");
    for (i = 0; (i < (sizeof(buffer)-1) && ((ch = fgetc(stream)) != EOF) &&
                        (ch != '\n')); i++)
         buffer[i] = ch;
    buffer[i] = '\0';
   if  (fclose(stream))
       perror("fclose  error");
    printf("The input line was : %s\n", buffer);
   return  0;

   /*************************************************************************
        If myfile.dat contains: one two three

        The output should be:

        The input line was : one two three
   *************************************************************************/
}
```

"_fgetchar — Read Single Character from **stdin**" on page 202
"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"fputc — Write Character" on page 227
"getc – getchar — Read a Character" on page 270

“_getch - _getche — Read Character from Keyboard” on page 272
“<stdio.h>” on page 774

_fgetchar — Read Single Character from stdin

Format

```
#include <stdio.h>
int _fgetchar(void);
```

Description

Language Level: Extension

_fgetchar reads a single character from the **stdin** stream. It is equivalent to the following fgetc call:

```
fgetc(c, stdin);
```

For portability, use the ANSI/ISO fgetc function instead of _fgetchar.

Return Value

_fgetchar returns the character read. A return value of EOF indicates an error or end-of-file position. Use feof or ferror to tell whether the return value indicates an error or an end-of-file position.

This example gathers a line of input from **stdin** using _fgetchar:

```
#include <stdio.h>

int main(void)
{
   char  buffer[81];
   int  i,ch;

    printf("Please input a line of characters...\n");
    for (i = 0; (i < 80) && ((ch = _fgetchar()) != EOF) && (ch != '\n'); i++)
        buffer[i] = ch;
    buffer[i] = '\0';
    printf("The input line was : %s\n", buffer);
   return  0;

   /*****************************************************************************
        The output should be:

        Please input a line of characters...
        This is a simple program.
        The input line was : This is a simple program.
   *****************************************************************************/
}
```

"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"fgetc — Read a Character" on page 200
"_fputchar — Write Character" on page 229
"getc – getchar — Read a Character" on page 270
"_getch - _getche — Read Character from Keyboard" on page 272
"<stdio.h>" on page 774

fgetpos — Get File Position

Format

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

Language Level: ANSI, SAA, XPG4

fgetpos stores the current position of the file pointer associated with *stream* into the object pointed to by *pos*. The value pointed to by *pos* can be used later in a call to fsetpos to reposition the *stream*.

Note: For buffered text streams, fgetpos returns an incorrect file position if the file contains new-line characters instead of carriage-return line-feed combinations. Your file would only contain new-line characters if you previously used it as a binary stream. To avoid this problem, either continue to process the file as a binary stream, or use unbuffered I/O operations.

Return Value

fgetpos returns 0 if successful. On error, fgetpos returns nonzero and sets errno to a nonzero value.

This example opens the file myfile.dat for reading and stores the current file pointer position into the variable *pos*.

```
#include <stdio.h>

FILE *stream;

int main(void)
{
   int  retcode;
   fpos_t  pos;

    stream = fopen("myfile.dat", "rb");

     /* The value returned by fgetpos can be used by fsetpos                    */
     /* to set the file pointer if 'retcode' is 0                              */

    if ( 0 == (retcode = fgetpos(stream, &pos)) )
        printf("Current position of file pointer found.\n");
   fclose(stream);
   return  0;

   /*****************************************************************************
        If myfile.dat exists, the output should be:

        Current position of file pointer found.
   *****************************************************************************/
}
```

"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"ftell — Get Current Position" on page 257

fgetpos

"<stdio.h>" on page 774

fgets — Read a String

Format

```
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fgets reads characters from the current *stream* position up to and including the first new-line character (\n), up to the end of the stream, or until the number of characters read is equal to *n*-1, whichever comes first. fgets stores the result in *string* and adds a null character (\0) to the end of the string. The *string* includes the new-line character, if read. If *n* is equal to 1, the *string* is empty.

Return Value

fgets returns a pointer to the *string* buffer if successful. A NULL return value indicates an error or an end-of-file condition. Use feof or ferror to determine whether the NULL value indicates an error or the end of the file. In either case, the value of the string is unchanged.

This example gets a line of input from a data stream. The example reads no more than MAX_LEN - 1 characters, or up to a new-line character, from the stream.

```
#include <stdio.h>

#define   MAX_LEN           100

int main(void)
{
   FILE  *stream;
   char  line[MAX_LEN],*result;

    stream = fopen("myfile.dat", "rb");
    if ((result = fgets(line, MAX_LEN, stream)) != NULL)
        printf("The string is %s\n", result);
   if  (fclose(stream))
       perror("fclose  error");
   return  0;

   /*************************************************************************
        If myfile.dat contains: This is my data file.

        The output should be:

        The string is This is my data file.
   *************************************************************************/
}
```

"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"fputs — Write String" on page 230
"gets — Read a Line" on page 281
"puts — Write a String" on page 441

fgets

"<stdio.h>" on page 774

fgetwc — Read Wide Character from Stream

Format

```
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

Language Level: ANSI 93

fgetwc reads the next multibyte character from the input stream pointed to by *stream*, converts it to a wide character, and advances the associated file position indicator for the stream (if defined).

The behavior of fgetwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

Using non-wide-character functions with fgetwc on the same stream results in undefined behavior.

After calling fgetwc, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling fgetwc.

Return Value

fgetwc returns the next wide character that corresponds to the multibyte character from the input stream pointed to by *stream*. If the stream is at EOF, the EOF indicator for the stream is set and fgetwc returns WEOF.

If a read error occurs, the error indicator for the stream is set and fgetwc returns WEOF. If an encoding error occurs (an error converting the multibyte character into a wide character), fgetwc sets **errno** to EILSEQ and returns WEOF.

Use ferror and feof to distinguish between a read error and an EOF. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

This example opens a file, reads in each wide character using fgetwc, and prints out the characters.

fgetwc

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE     *stream;
    wint_t  wc;

     if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
          printf("Unable to open: \"fgetwc.dat\"\n");
          exit(1);
    }

     errno = 0;
     while (WEOF != (wc = fgetwc(stream)))
          printf("wc = %lc\n", wc);

     if (EILSEQ == errno) {
          printf("An invalid wide character was encountered.\n");
          exit(1);
    }
    fclose(stream);
    return  0;

    /*********************************************************************
          Assuming the file fgetwc.dat contains:

          Hello  world!

          The output should be similar to:

          wc = H
          wc = e
          wc = l
          wc = l
          wc = o
          :
    *********************************************************************/
}
```

"fgetc — Read a Character" on page 200
"_fgetchar — Read Single Character from **stdin**" on page 202
"fgetws — Read Wide-Character String from Stream" on page 209
"fputwc — Write Wide Character" on page 232
"_getch - _getche — Read Character from Keyboard" on page 272
"<stdio.h>" on page 774
"<wchar.h>" on page 780

fgetws — Read Wide-Character String from Stream

Format

#include <wchar.h>
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);

Description

Language Level: ANSI 93, XPG4

fgetws reads wide characters from the *stream* into the array pointed to by *wcs*. At most, *n* − 1 wide characters are read. fgetws stops reading characters after WEOF, or after it reads a new-line wide character (which is retained). It adds a null wide character immediately after the last wide character read into the array.

fgetws advances the file position unless there is an error. If an error occurs, the file position is undefined.

The behavior of fgetws is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

Using non-wide-character functions with fgetws on the same stream results in undefined behavior.

After calling fgetws, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless WEOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling fgetws.

Return Value

If successful, fgetws returns a pointer to the wide-character string *wcs*. If WEOF is encountered before any wide characters have been read into *wcs*, the contents of *wcs* remain unchanged and fgetws returns a null pointer. If WEOF is reached after data has already been read into the string buffer, fgetws returns a pointer to the string buffer to indicate success. A subsequent call would return NULL because WEOF would be reached without any data being read.

If a read error occurs, the contents of *wcs* are indeterminate and fgetws returns NULL. If an encoding error occurs (in converting a wide character to a multibyte character), fgetws sets **errno** to EILSEQ and returns NULL.

If *n* equals 1, the *wcs* buffer has only room for the terminating null character and nothing is read from the stream. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless the buffer is flushed or the stream pointer repositioned first.)

If *n* is greater than 1, fgetws fails only if an I/O error occurs or if WEOF is reached before data is read from the stream. Use ferror and feof to distinguish between a read error and a WEOF. WEOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the WEOF indicator.

This example opens a file, reads in the file contents using fgetws, then prints the file contents.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE       *stream;
    wchar_t    wcs[100];

     if (NULL == (stream = fopen("fgetws.dat", "r"))) {
          printf("Unable to open: \"fgetws.dat\"\n");
          exit(1);
    }

     errno = 0;
     if (NULL == fgetws(wcs, 100, stream)) {
          if (EILSEQ == errno) {
               printf("An invalid wide character was encountered.\n");
               exit(1);
          }
          else if (feof(stream))
                    printf("End of file reached.\n");
               else
                    perror("Read  error.\n");
    }
     printf("wcs = \"%ls\"\n", wcs);
    fclose(stream);
    return  0;

    /*********************************************************************
          Assuming the file fgetws.dat contains:

          This test string should not return -1

          The output should be similar to:

          wcs = "This test string should not return -1"
    *********************************************************************/
}
```

"fgets — Read a String" on page 205
"fgetwc — Read Wide Character from Stream" on page 207
"fputws — Write Wide-Character String" on page 234.
"<stdio.h>" on page 774
"<wchar.h>" on page 780

_filelength — Determine File Length

Format

#include <io.h>
long _filelength(int *handle*);

Description

Language Level: Extension

_filelength returns the length, in bytes, of the file associated with *handle*. The length of the file will be correct even if you have the handle opened and have appended data to the file.

Return Value

A return value of –1L indicates an error, and errno is set to one of the following values:

Value	Meaning
EBADF	The file handle is incorrect or the mode specified does not match the mode you opened the file with.
EOS2ERR	The call to the operating system was not successful.

This example opens a file and tries to determine the current length of the file using _filelength.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
   long  length;
   int  fh;

   printf("\nCreating  sample.dat.\n");
    system("echo Sample Program > sample.dat");
    if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
         printf("Unable to open sample.dat.\n");
        return  EXIT_FAILURE;
   }
    if (-1 == (length = _filelength(fh))) {
         printf("Unable to determine length of sample.dat.\n");
        return  EXIT_FAILURE;
   }
    printf("Current length of sample.dat is %d.\n", length);
   close(fh);
   return  0;

   /*************************************************************************
         The output should be:

        Creating  sample.dat.
         Current length of sample.dat is 17.
   *************************************************************************/
}
```

_filelength

"_chsize — Alter Length of File" on page 85
"__eof — Determine End of File" on page 171
"<io.h>" on page 764

fileno — Determine File Handle

Format

```
#include <stdio.h>
int fileno(FILE *stream);
```

Description

Language Level: XPG4, Extension

fileno determines the file handle currently associated with *stream*.

Note: In earlier releases of VisualAge C++, fileno began with an underscore (_fileno). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _fileno to fileno for you.

Return Value

fileno returns the file handle. If the function fails, the return value is -1 and the **errno** variable may be set to one of the following values:

Value	Meaning
ENULLFCB	The input stream is NULL.
EBADTYPE	The input stream file is not a stream file.

The result is undefined if *stream* does not specify an open file.

This example determines the file handle of the **stderr** data stream.

```
#include <stdio.h>

int main(void)
{
   int  result;

    result = 0xFFFF&fileno(stderr);
    printf("The file handle associated with stderr is %d.\n", result);
   return  0;

   /*****************************************************************************
       The output should be similar to:

       The file handle associated with stderr is 2.
   *****************************************************************************/
}
```

"dup — Associate Second Handle with Open File" on page 160
"dup2 — Associate Second Handle with Open File" on page 163
"fopen — Open Files" on page 218
"freopen — Redirect Open Files" on page 242
"isatty — Test Handle for Character Device" on page 321
"open — Open File" on page 418
"<stdio.h>" on page 774

floor — Integer <= Argument

Format

```
#include <math.h>
double floor(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

floor calculates the largest integer that is less than or equal to *x*.

Return Value

floor returns the floating-point result as a double value.

The result of floor cannot have a range error.

This example assigns *y* value of the largest integer less than or equal to 2.8 and *z* the value of the largest integer less than or equal to -2.8.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  y,z;

    y = floor(2.8);
    z = floor(-2.8);
    printf("floor(    2.8 ) = %lf\n", y);
    printf("floor( -2.8 ) = %lf\n", z);
   return  0;

   /**************************************************************************
        The output should be:

        floor(    2.8 ) = 2.000000
        floor( -2.8 ) = -3.000000
   **************************************************************************/
}
```

"ceil — Find Integer >= Argument" on page 77
"fmod — Calculate Floating-Point Remainder" on page 217
"<math.h>" on page 770

_flushall — Write Buffers to Files

Format

```
#include <stdio.h>
int _flushall(void);
```

Description

Language Level: Extension

_flushall causes the system to write to file the contents of all buffers associated with open output streams (including **stdin**, **stdout**, and **stderr**). It clears all buffers associated with open input streams of their current contents. The next read operation, if there is one, reads new data from the input files into the buffers. All streams remain open after the call.

For portability, use the ANSI/ISO function fflush instead of _flushall.

Return Value

_flushall returns the number of open streams of input and output. If an error occurs, _flushall returns EOF.

In this example, _flushall completes any pending input or output on all streams by flushing all buffers.

```
#include <stdio.h>

int main(void)
{
    int  i,numflushed;
     char buffer1[5] =    { 1,2,3,4 };
     char buffer2[5] =    { 5,6,7,8 };
     char *file1 = "file1.dat";
     char *file2 = "file2.dat";
    FILE  *stream1,*stream2;

     stream1 = fopen(file1, "a+");
     stream2 = fopen(file2, "a+");
     for (i = 0; i <= sizeof(buffer1); i++) {
         fputc(buffer1[i],  stream1);
         fputc(buffer2[i],  stream2);
    }
     numflushed = _flushall();                          /* all streams flushed              */
     printf("Number of files flushed = %d\n", numflushed);
    fclose(stream1);
    fclose(stream2);
    return  0;

    /*************************************************************************
         The output should be:

         Number of files flushed = 5
    *************************************************************************/
}
```

_flushall

"close — Close File Associated with Handle" on page 92
"fclose — Close Stream" on page 187
"fflush — Write Buffer to File" on page 199
"<stdio.h>" on page 774

fmod — Calculate Floating-Point Remainder

Format

```
#include <math.h>
double fmod(double x, double y);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fmod calculates the floating-point remainder of *x*/*y*. The absolute value of the result is always less than the absolute value of *y*. The result will have the same sign as *x*.

Return Value

fmod returns the floating-point remainder of *x*/*y*. If *y* is zero or if *x*/*y* causes an overflow, fmod returns 0.

This example computes *z* as the remainder of *x*/*y*; here, *x*/*y* is -3 with a remainder of -1.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y,z;

    x = -10.0;
    y = 3.0;
    z = fmod(x, y);                                          /* z = -1.0          */
    printf("fmod( %lf, %lf) = %lf\n", x, y, z);
   return  0;

   /*************************************************************************
        The output should be:

        fmod( -10.000000, 3.000000) = -1.000000
   *************************************************************************/
}
```

"ceil — Find Integer >= Argument" on page 77
"fabs — Calculate Floating-Point Absolute Value" on page 182
"floor — Integer <= Argument" on page 214
"<math.h>" on page 770

fopen — Open Files

Format

#include <stdio.h>
FILE *fopen(const char **filename*, const char **mode*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

fopen opens the file specified by *filename*. *mode* is a character string specifying the type of access requested for the file. The *mode* variable contains one positional parameter followed by optional keyword parameters.

The possible values for the positional parameters are:

Mode	Description
r	Open a text file for reading. The file must exist.
w	Create a text file for writing. If the given file exists, its contents are destroyed.
a	Open a text file in append mode for writing at the end of the file. fopen creates the file if it does not exist.
r+	Open a text file for both reading and writing. The file must exist.
w+	Create a text file for both reading and writing. If the given file exists, its contents are destroyed.
a+	Open a text file in append mode for reading or updating at the end of the file. fopen creates the file if it does not exist.
rb	Open a binary file for reading. The file must exist.
wb	Create an empty binary file for writing. If the file exists, its contents are destroyed.
ab	Open a binary file in append mode for writing at the end of the file. fopen creates the file if it does not exist.
r+b *or* rb+	Open a binary file for both reading and writing. The file must exist.
w+b *or* wb+	Create an empty binary file for both reading and writing. If the file exists, its contents will be destroyed.
a+b *or* ab+	Open a binary file in append mode for writing at the end of the file. fopen creates the file if it does not exist.

Warning: Use the w, w+, wb, w+b, and wb+ parameters with care; data in existing files of the same name will be lost.

Text files contain printable characters and control characters organized into lines. Each line ends with a new-line character, except for the last line, which does not require one. The system may insert or convert control characters in an output text stream.

Note: Data output to a text stream may not compare as equal to the same data on input.

Binary files contain a series of characters. For binary files, the system does not translate control characters on input or output.

When you open a file with a, a+, ab, a+b or ab+ mode, all write operations take place at the end of the file. Although you can reposition the file pointer using fseek or rewind, the write functions move the file pointer back to the end of the file before they carry out any operation. This action prevents you from overwriting existing data.

When you specify the update mode (using + in the second or third position), you can both read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as fseek, fsetpos, rewind, or fflush. Output may immediately follow input if the end-of-file was detected.

The keyword parameters are:

blksize=*value* Specifies the maximum length, in bytes, of a physical block of records. For fixed-length records, the maximum size is 32760 bytes. For variable-length records, the maximum is 32756. The default buffer size is 4096 bytes.

lrecl=*value* Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records. For fixed-length records, the maximum length is 32760 bytes. For variable-length records, the maximum is 32756. If the value of LRECL is larger than the value of BLKSIZE, the LRECL value is ignored.

recfm=*value* *value* can be:

F fixed-length, unblocked records
V variable-length, unblocked records

The default for the VisualAge for C++ compiler is fixed-length record format.

type=*value* *value* can be:

memory This parameter identifies this file as a memory file that is accessible only from C programs. If you want to use a memory file, you must compile with the /Sv option.

The VisualAge for C++ compiler does not support record I/O.

fopen generally fails if parameters are mismatched.

The file attributes can be altered only if the open mode specified with the fopen function is one of the write modes (w, w+, wb, or wb+). The system deletes the existing file and creates a new file with the attributes specified in fopen.

Return Value

fopen returns a pointer to a file structure that can be used to access the open file. A NULL pointer return value indicates an error.

This example attempts to open a file for reading.

```
#include <stdio.h>

int main(void)
{
   FILE  *stream;

    if (NULL == (stream = fopen("myfile.dat", "r")))
        printf("Could not open data file\n");
   else
        fclose(stream);

     /* The following call opens a fixed record length file                    */
     /* for reading and writing in record mode.                                */

    if (NULL == (stream = fopen("myfile.dat",
                                "rb+, lrecl=80, blksize=240, recfm=f, type=record")))
        printf("Could not open data file\n");
   else
        fclose(stream);
   return  0;

   /*****************************************************************************
        The output should be:

        Could not open data file
   *****************************************************************************/
}
```

"creat — Create New File" on page 100
"fclose — Close Stream" on page 187
"fflush — Write Buffer to File" on page 199
"fread — Read Items" on page 236
"freopen — Redirect Open Files" on page 242
"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"fwrite — Write Items" on page 263
"rewind — Adjust Current File Position" on page 465
"<stdio.h>" on page 774

_fpatan — Calculate Arctangent

Format

```
#include <builtin.h>
double _fpatan(double x);
```

Description

Language Level: Extension

_fpatan calculates the arctangent of *x*, which is a value in radians between -π/2 and π/2. This function causes the compiler to emit the appropriate 80387 instruction for the calculation of arctangent.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fpatan.
You cannot parenthesize a _fpatan function call, because parentheses specify a call to the backing code for the function.

Return Value

This function returns the arctangent of *x*.

This example calculates the arctangent of *x*.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  x;

    x = 0.45;
    printf("The arctangent of %lf is %lf.\n", x, _fpatan(x));
   return  0;

   /*****************************************************************************
        The output should be :

        The arctangent of 0.450000 is 0.422854.
   *****************************************************************************/
}
```

"atan – atan2 — Calculate Arctangent" on page 53
"_fptan — Calculate Tangent" on page 226
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<builtin.h>" on page 761

_fpreset — Reset Floating-Point Unit

Format

```
#include <float.h>
void _fpreset(void);
```

Description

Language Level: Extension

_fpreset resets the floating-point unit to the default state that the math library requires to function correctly. The value of this default state may change between releases of VisualAge for C++. You can determine the default state by calling _fpreset and then calling _control87 with 0 as the parameter.

This function is often used within signal handlers. If a program traps floating-point error signals (SIGFPE) with signal, the program can safely recover from floating-point errors by calling longjmp. For more information on signal handlers, see "Signal and Windows Exception Handling" the *Programming Guide*.

_fpreset resets the floating-point unit of the current thread only. It does not affect any other threads that may be processing.

Return Value

There is no return value.

This example establishes the function fphandler as a floating-point error handler. The main program produces a floating-point error, which causes control to be passed to fphandler. The fphandler function calls _fpreset to reset the floating-point unit, and then returns to main.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <float.h>
```

```
jmp_buf mark;

void fphandler(int sig)
{
     printf("Floating point signal = %d\n", sig);
     _fpreset();                          /* Reinitialize floating-point package                  */
    longjmp(mark,  -1);
}

int main(void)
{
     double a = 1.0,b = 0.0,c;

     if (SIG_ERR == signal(SIGFPE, (_SigFunc)fphandler))
          return  EXIT_FAILURE;
     if (0 == setjmp(mark)) {
          c = a/b;                                /* generate floating-point error                    */
          printf("Should never get here\n");
     }
     printf("Recovered from floating-point error\n");
    return  0;

   /*****************************************************************************
          The output should be:

          Floating point signal = 3
          Recovered from floating-point error
   *****************************************************************************/
}
```

"_clear87 — Clear Floating-Point Status Word" on page 88
"_control87 — Set Floating-Point Control Word" on page 94
"longjmp — Restore Stack Environment" on page 360
"signal — Handle Interrupt Signals" on page 510
"_status87 — Get Floating-Point Status Word" on page 534
"<float.h>" on page 763

fprintf — Write Formatted Data to a Stream

Format

#include <stdio.h>
int fprintf(FILE **stream*, const char **format-string*, *argument-list*);

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

fprintf formats and writes a series of characters and values to the output *stream*. fprintf converts each entry in *argument-list*, if any, and writes to the stream according to the corresponding format specification in the *format-string*.

The *format-string* has the same form and function as the *format-string* argument for printf. See "printf — Print Formatted Characters" on page 429 for a description of the *format-string* and the argument list.

In extended mode, fprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See "Infinity and NaN Support" on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, fprintf prints (null). (In previous releases of VisualAge C++, printf produced no output for a null string.)

Return Value

fprintf returns the number of bytes printed or a negative value if an output error occurs.

This example sends a line of asterisks for each integer in the array count to the file myfile. The number of asterisks printed on each line corresponds to an integer in the array.

```
#include <stdio.h>

int count[10] =   { 1, 5, 8, 3, 0, 3, 5, 6, 8, 10 } ;

int main(void)
{
   int  i,j;
   FILE  *stream;
```

```
    stream = fopen("myfile.dat", "w");

                        /* Open the stream for writing                          */

    for (i = 0; i < sizeof(count)/sizeof(count[0]); i++) {
         for (j = 0; j < count[i]; j++)
             fprintf(stream,  "*");

                        /* Print asterisk                                       */

         fprintf(stream,  "\n");

                        /* Move to the next line                                */

    }
    fclose(stream);
    return  0;

    /*************************************************************************
         The output data file myfile.dat should contain:

         *
         *****
         ********
         ***

         ***
         *****
         ******
         ********
         **********
    *************************************************************************/
}
```

"_cprintf — Print Characters to Screen" on page 98
"fscanf — Read Formatted Data" on page 245
"printf — Print Formatted Characters" on page 429
"sprintf — Print Formatted Data to Buffer" on page 525
"vfprintf — Print Argument Data to Stream" on page 698
"vprintf — Print Argument Data" on page 700
"vsprintf — Print Argument Data to Buffer" on page 702
"vswprintf — Format and Write Wide Characters to Buffer" on page 704
"Infinity and NaN Support" on page 27
"<stdio.h>" on page 774

_fptan — Calculate Tangent

Format

```
#include <builtin.h>
double _fptan(double x);
```

Description

Language Level: Extension

_fptan calculates the tangent of *x*, where *x* is expressed in radians. The value of *x* must be less than 2**63. This function causes the compiler to emit the appropriate 80387 instruction for the calculation of tangent.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fptan.
You cannot parenthesize a _fptan function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

_fptan returns the tangent of *x* expressed in radians.

This example calculates the tangent of pi/4.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  pi;

    pi = 3.1415926;
    printf("The tangent of %lf is %lf.\n", pi/4.0, _fptan(pi/4.0));
   return  0;

   /*****************************************************************************
        The output should be :

        The tangent of 0.785398 is 1.000000.
   *****************************************************************************/
}
```

"atan – atan2 — Calculate Arctangent" on page 53
"_fpatan — Calculate Arctangent" on page 221
"tan — Calculate Tangent" on page 617
"tanh — Calculate Hyperbolic Tangent" on page 618
"<builtin.h>" on page 761

fputc — Write Character

Format

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fputc converts *c* to an unsigned char and then writes *c* to the output *stream* at the current position and advances the file position appropriately. If the stream is opened with one of the append modes, the character is appended to the end of the stream.

fputc is identical to putc except that it is never replaced by a macro.

Return Value

fputc returns the character written. A return value of EOF indicates an error.

This example writes the contents of buffer to a file called myfile.dat.

Note: Because the output occurs as a side effect within the second expression of the for statement, the statement body is null.

```
#include <stdio.h>
#include <stdlib.h>

#define   NUM_ALPHA        26

int main(void)
{
   FILE  *stream;
   int  i;
   int  ch;

    /* Don't forget the NULL char at the end of the string!                    */

    char buffer[NUM_ALPHA+1] = "abcdefghijklmnopqrstuvwxyz";

    if ((stream = fopen("myfile.dat", "w")) != NULL) {

       /* Put buffer into file                                                 */

         for (i = 0; (i < sizeof(buffer)) && ((ch = fputc(buffer[i], stream)) !=
            EOF);  ++i)
            ;
#ifdef _M_PPC
         fputc( '\n', stream );
#endif
```

```
    fclose(stream);
    return  0;
}
else
     perror("Error opening myfile.dat");
return  EXIT_FAILURE;

/*************************************************************************
     The output data file myfile.dat should contain:

    abcdefghijklmnopqrstuvwxyz
*************************************************************************/
}
```

"fgetc — Read a Character" on page 200
"_fputchar — Write Character" on page 229
"putc – putchar — Write a Character" on page 436
"_putch — Write Character to Screen" on page 438
"<stdio.h>" on page 774

_fputchar — Write Character

Format

```
#include <stdio.h>
int _fputchar(int c);
```

Description

Language Level: Extension

_fputchar writes the single character *c* to the stdout stream at the current position. It is equivalent to the following fputc call:

```
fputc(c, stdout);
```

For portability, use the ANSI/ISO fputc function instead of _fputchar.

Return Value

_fputchar returns the character written. A return value of EOF indicates that a write error has occurred. Use ferror and feof to tell whether this is an error condition or the end of the file.

This example writes the contents of buffer to stdout:

```
#include <stdio.h>

int main(void)
{
   char  buffer[80];
    int i,ch = 1;

    for (i = 0; i < 80; i++)
        buffer[i] = 'c';
    for (i = 0; (i < 80) && (ch != EOF); i++)
        ch = _fputchar(buffer[i]);
   printf("\n");
   return  0;

   /*****************************************************************************
        The output should be similar to:

   ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
   *****************************************************************************/
}
```

"_fgetchar — Read Single Character from **stdin**" on page 202
"fputc — Write Character" on page 227
"putc – putchar — Write a Character" on page 436
"_putch — Write Character to Screen" on page 438
"<stdio.h>" on page 774

fputs — Write String

Format

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fputs copies *string* to the output *stream* at the current position. It does not copy the null character (\0) at the end of the string.

Return Value

fputs returns EOF if an error occurs; otherwise, it returns a non-negative value.

This example writes a string to a stream.

```
#include <stdio.h>

#define   NUM_ALPHA        26

int main(void)
{
   FILE  *stream;
   int  num;

    /* Do not forget that the '\0' char occupies one character                    */

    static char buffer[NUM_ALPHA+1] = "abcdefghijklmnopqrstuvwxyz";

    if ((stream = fopen("myfile.dat", "w")) != NULL) {

        /* Put buffer into file                                                   */

        if ((num = fputs(buffer, stream)) != EOF) {

           /* Note that fputs() does not copy the \0 character                   */

            printf("Total number of characters written to file = %i\n", num);
           fclose(stream);
        }
        else                                                  /* fputs failed     */
           perror("fputs  failed");
   }
   else
        perror("Error opening myfile.dat");
   return  0;

   /*************************************************************************
        The output should be:

        Total number of characters written to file = 26
   *************************************************************************/
}
```

"_cputs — Write String to Screen" on page 99
"fgets — Read a String" on page 205

“gets — Read a Line” on page 281
“puts — Write a String” on page 441
“<stdio.h>” on page 774

fputwc — Write Wide Character

Format

#include <wchar.h>
wint_t fputwc(wint_t *wc*, FILE **stream*);

Description

Language Level: ANSI 93, XPG4

fputwc converts the wide character *wc* to a multibyte character and writes it to the output stream pointed to by *stream* at the current position. It also advances the file position indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the stream.

The behavior of fputwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur. Using non-wide-character functions with fputwc on the same stream results in undefined behavior.

After calling fputwc, flush the buffer or reposition the stream pointer before calling a read function for the stream. After reading from the stream, flush the buffer or reposition the stream pointer before calling fputwc, unless EOF has been reached.

Return Value

fputwc returns the wide character written. If a write error occurs, the error indicator for the stream is set and fputwc returns WEOF. If an encoding error occurs during conversion from wide character to a multibyte character, fputwc sets **errno** to EILSEQ and returns WEOF.

This example opens a file and uses fputwc to write wide characters to the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>
```

```
int main(void)
{
    FILE       *stream;
     wchar_t *wcs = L"A character string.";
    int         i;

     if (NULL == (stream = fopen("fputwc.out", "w"))) {
          printf("Unable to open: \"fputwc.out\".\n");
         exit(1);
    }

     for (i = 0; wcs[i] != L'\0'; i++) {
          errno = 0;
          if (WEOF == fputwc(wcs[i], stream)) {
               printf("Unable to fputwc() the wide character.\n"
                         "wcs[%d] = 0x%lx\n", i, wcs[i]);
               if (EILSEQ == errno)
                    printf("An invalid wide character was encountered.\n");
              exit(1);
          }
    }
    fclose(stream);
    return  0;

    /*****************************************************************************
          The output file fputwc.out should contain :

          A character string.
    *****************************************************************************/
}
```

“fgetwc — Read Wide Character from Stream” on page 207
“fputc — Write Character” on page 227
“_fputchar — Write Character” on page 229
“fputws — Write Wide-Character String” on page 234
“_putch — Write Character to Screen” on page 438
“<stdio.h>” on page 774
“<wchar.h>” on page 780

fputws — Write Wide-Character String

Format

```
#include <wchar.h>
int fputws(const wchar_t *wcs, FILE *stream);
```

Description

Language Level: ANSI, SAA, XPG4

fputws converts the wide-character string *wcs* to a multibyte-character string and writes it to *stream* as a multibyte character string. It does not write the terminating null byte.

The behavior of fputws is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur. Using non-wide-character functions with fputws on the same stream results in undefined behavior.

After calling fputws, flush the buffer or reposition the stream pointer before calling a read function for the stream. After a read operation, flush the buffer or reposition the stream pointer before calling fputws, unless EOF has been reached.

Return Value

fputws returns a non-negative value if successful. If a write error occurs, the error indicator for the stream is set and fputws returns −1. If an encoding error occurs in converting the wide characters to multibyte characters, fputws sets **errno** to EILSEQ and returns −1.

This example opens a file and writes a wide-character string to the file using fgetws.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE       *stream;
    wchar_t *wcs = L"This test string should not return -1";

    if (NULL == (stream = fopen("fputws.out", "w"))) {
        printf("Unable to open: \"fputws.out\".\n");
       exit(1);
   }

    errno = 0;
    if (EOF == fputws(wcs, stream)) {
        printf("Unable to complete fputws() function.\n");
        if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
       exit(1);
   }
   fclose(stream);
   return  0;
```

```
/*****************************************************************************
    The output file fputws.out should contain :

    This test string should not return -1
*****************************************************************************/
}
```

"fgetws — Read Wide-Character String from Stream" on page 209
"fputs — Write String" on page 230
"fputwc — Write Wide Character" on page 232
"<stdio.h>" on page 774
"<wchar.h>" on page 780

fread — Read Items

Format

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count,
                           FILE  *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fread reads up to *count* items of *size* length from the input *stream* and stores them in the given *buffer*. The position in the file increases by the number of bytes read.

Return Value

fread returns the number of full items successfully read, which can be less than *count* if an error occurs or if the end-of-file is met before reaching *count*. If *size* or *count* is 0, fread returns zero and the contents of the array and the state of the stream remain unchanged.

Use ferror and feof to distinguish between a read error and an end-of-file.

This example attempts to read NUM_ALPHA characters from the file myfile.dat. If there are any errors with either fread or fopen, a message is printed.

```
#include <stdio.h>

#define   NUM_ALPHA        26

int main(void)
{
    FILE  *stream;
     int num;                              /* number of characters read from stream          */

     /* Do not forget that the '\0' char occupies one character too!                        */

    char  buffer[NUM_ALPHA+1];

     if ((stream = fopen("myfile.dat", "r")) != NULL) {
          num = fread(buffer, sizeof(char), NUM_ALPHA, stream);
          if (num) {                                                    /* fread success       */
               printf("Number of characters has been read = %i\n", num);
               buffer[num] = '\0';
               printf("buffer = %s\n", buffer);
              fclose(stream);
          }
```

```
        else {                                                  /* fread failed          */
            if (ferror(stream))                                 /* possibility 1        */
                perror("Error reading myfile.dat");
            else
                if (feof(stream))                               /* possibility 2        */

                    perror("EOF  found");
        }
    }
    else
        perror("Error opening myfile.dat");
    return  0;

    /*****************************************************************************
        The output should be:

        Number of characters has been read = 26
        buffer = abcdefghijklmnopqrstuvwxyz
    *****************************************************************************/
}
```

"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"fopen — Open Files" on page 218
"fwrite — Write Items" on page 263
"read — Read Into Buffer" on page 450
"<stdio.h>" on page 774

free — Release Storage Blocks

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void free(void *ptr);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

free frees a block of storage. *ptr* points to a block previously reserved with a call to one of the memory allocation functions (such as calloc, _umalloc, or _trealloc). The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of realloc) the block of storage. If *ptr* is NULL, free simply returns.

Note: Attempting to free a block of storage not allocated with a C memory management function or a block that was previously freed can affect the subsequent reserving of storage and lead to undefined results.

Return Value

There is no return value.

This example uses calloc to allocate storage for *x* array elements and then calls free to free them.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long *array;                          /* start of the array                */
    int num = 100;                        /* number of entries of the array    */

    printf("Allocating memory for %d long integers.\n", num);

    /* allocate num entries                                                     */

    if (NULL != (array = (long *)calloc(num, sizeof(long)))) {

      /*.....................................................................*/
       /*    do something with the array                                      */
      /*.....................................................................*/

        free(array);                                    /* deallocates array*/
    }
```

```
    else {                                                  /* Out of storage     */
        perror("Error: out of storage");
        abort();
    }
    return  0;

    /*****************************************************************************
        The output should be:

        Allocating memory for 100 long integers.
    *****************************************************************************/
}
```

Managing Memory in the *Programming Guide*
"_alloca — Temporarily Reserve Storage Block" on page 45
"calloc — Reserve and Initialize Storage" on page 75
"_debug_free — Release Memory" on page 126
"malloc — Reserve Storage Block" on page 376
"realloc — Change Reserved Storage Block Size" on page 452
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_freemod — Free User DLL

Format

#include <stdlib.h>
int _freemod(unsigned long *module_handle*);

Description

Language Level: Extension

_freemod frees all references to a dynamic link library (DLL) for the calling process. It is the counterpart of the _loadmod function, which loads a DLL for the process. The *module_handle* is the module handle of the DLL, which was returned by _loadmod.

Note: _loadmod and _freemod perform exactly the same function as the Windows APIs LoadLibrary and FreeLibrary. They are provided for compatibility with earlier VisualAge for C++ releases only, and will not be supported in future versions. Use LoadLibrary and FreeLibrary instead. For more details on these APIs, see the *Win32 Programmer's Reference*.

Return Value

_freemod returns 0 if successful. If not, _freemod returns -1 and sets **errno** to EOS2ERR.

This example loads the DLL mark with _loadmod, and uses the API GetProcAddress to get the address of the function dor from within the DLL. It then calls that function, which multiplies two integers passed to it. When the function and DLL are no longer needed, the program frees the DLL module. The types HMODULE and FARPROC are defined by including <windows.h>.

Note: To run this program, you must first create mark.dll. Copy the code for the PLUS function into a file called mark.c, and the code for the .DEF file into mark.def. Eliminate the comments from these two files. Compile with the command:

```
icc /Ge- mark.c mark.def
```

You can then run the example.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

typedef (_Optlink *fptr)(int, int);

int main(void) {

    int x = 4, y = 7;
    unsigned long handle;
    char *modname = "MARK";                          /* DLL name                  */
    char *fname = "PLUS";                            /* function name             */
    fptr faddr;                                      /* pointer to function    */

    /*    dynamically load the 'mark' DLL                                        */

    if (_loadmod(modname, &handle)) {
        printf("Error loading module %s\n", modname);
       return  EXIT_FAILURE;
    }

    /* get function address from DLL                 */

    faddr = (fptr)GetProcAddress((HMODULE)handle, fname);
    if (NULL != faddr) {
        printf("Calling the function from the %s DLL to add %d and %d\n",
                modname, x, y);
        printf("The result from the function call is %d\n", faddr(x, y));
    }
    else  {
        DWORD rc = GetLastError();
        printf("Error locating address of function %s, rc=%d\n", fname, rc);
       _freemod(handle);
       return  EXIT_FAILURE;
    }

    if (_freemod(handle)) {
        printf("Error in freeing the %s DLL module\n", modname);
       return  EXIT_FAILURE;
    }

    printf("Reference to the %s DLL module has been freed\n", modname);

   /*****************************************************************************
          The output should be:

          Calling the function from MARK DLL to add 4 and 7
          The result of the function call is 11
          Reference to the MARK DLL module has been freed
   *****************************************************************************/
}
```

"Building Dynamic Link Libraries" in the *Programming Guide*
"_loadmod — Load DLL" on page 348
"<stdlib.h>" on page 775

freopen — Redirect Open Files

Format

#include <stdio.h>
FILE *freopen(const char **filename*, const char **mode*, FILE **stream*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

freopen closes the file currently associated with *stream* and reassigns *stream* to the file specified by *filename*. The freopen function opens the new file associated with *stream* with the given *mode*, which is a character string specifying the type of access requested for the file. You can also use the freopen function to redirect the standard stream files **stdin**, **stdout**, and **stderr** to files that you specify.

If *filename* is an empty string, freopen closes and reopens the stream to the new open mode, rather than reassigning it to a new file or device. You can use freopen with no file name specified to change the mode of a standard stream from text to binary without redirecting the stream, for example:

```
fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text.

Portability Note: This method is included in the SAA C definition, but not in the ANSI/ISO C standard.

See "fopen — Open Files" on page 218 for a description of the *mode* parameter.

Return Value

freopen returns a pointer to the newly opened stream. If an error occurs, freopen closes the original file and returns a NULL pointer value.

This example closes the stream1 data stream and reassigns its stream pointer. Note that stream1 and stream2 will have the same value, but they will not necessarily have the same value as stream.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE  *stream,*stream1,*stream2;

    stream = fopen("myfile.dat", "r");
    stream1 = stream;
    if (NULL == (stream2 = freopen("", "w+", stream1)))
        return  EXIT_FAILURE;
   fclose(stream2);
   return  0;
}
```

"close — Close File Associated with Handle" on page 92

"fclose — Close Stream" on page 187
"_fcloseall — Close All Open Streams" on page 188
"fopen — Open Files" on page 218
"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"<stdio.h>" on page 774

frexp — Separate Floating-Point Value

Format

```
#include <math.h>
double frexp(double x, int *expptr);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

frexp breaks down the floating-point value *x* into a term *m* for the mantissa and another term *n* for the exponent, such that $x=m*2^n$, and the absolute value of *m* is greater than or equal to 0.5 and less than 1.0 or equal to 0. frexp stores the integer exponent *n* at the location to which *expptr* points.

Return Value

frexp returns the mantissa term *m*. If *x* is 0, frexp returns 0 for both the mantissa and exponent. The mantissa has the same sign as the argument *x*. The result of the frexp function cannot have a range error.

This example decomposes the floating-point value of *x*, 16.4, into its mantissa 0.5125, and its exponent 5. It stores the mantissa in *y* and the exponent in *n*.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,m;
   int  n;

    x = 16.4;
    m = frexp(x, &n);
    printf("The mantissa is %lf and the exponent is %d\n", m, n);
   return  0;

   /*****************************************************************************
        The output should be:

        The mantissa is 0.512500 and the exponent is 5
   *****************************************************************************/
}
```

"ldexp — Multiply by a Power of Two" on page 340
"modf — Separate Floating-Point Value" on page 412
"<math.h>" on page 770

fscanf — Read Formatted Data

Format

#include <stdio.h>
int fscanf (FILE *_stream_, const char *_format-string_, _argument-list_);

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

fscanf reads data from the current position of the specified _stream_ into the locations given by the entries in _argument-list_, if any. Each entry in _argument-list_ must be a pointer to a variable with a type that corresponds to a type specifier in _format-string_.

The _format-string_ controls the interpretation of the input fields and has the same form and function as the _format-string_ argument for the scanf function. See “scanf — Read Data” on page 486 for a description of _format-string_.

In extended mode, the fscanf function also reads in the strings "INFINITY", "INF", and "NAN" (in upper or lowercase) and converts them to the corresponding floating-point value. The sign of the value is determined by the format specification. See “Infinity and NaN Support” on page 27 for more information on infinity and NaN values.

Return Value

fscanf returns the number of fields that it successfully converted and assigned. The return value does not include fields that fscanf read but did not assign.

The return value is EOF if an input failure occurs before any conversion, or the number of input items assigned if successful.

This example opens the file myfile.dat for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.

fscanf

```
#include <stdio.h>

#define   MAX_LEN            80

int main(void)
{
    FILE  *stream;
    long  l;
    float  fp;
    char  s[MAX_LEN+1];
    char  c;

     stream = fopen("myfile.dat", "r");

       /* Put in various data.                                                    */

     fscanf(stream, "%s", &s[0]);
     fscanf(stream, "%ld", &l);
     fscanf(stream, "%c", &c);
     fscanf(stream, "%f", &fp);
     printf("string = %s\n", s);
     printf("long double = %ld\n", l);
     printf("char = %c\n", c);
     printf("float = %f\n", fp);
    return  0;

    /*************************************************************************
          If myfile.dat contains:
          abcdefghijklmnopqrstuvwxyz  343.2.

          The output should be:

          string = abcdefghijklmnopqrstuvwxyz
          long double = 343
          char = .
          float = 2.000000
    *************************************************************************/
}
```

"Infinity and NaN Support" on page 27
"_cscanf — Read Data from Keyboard" on page 112
"fprintf — Write Formatted Data to a Stream" on page 224
"scanf — Read Data" on page 486
"sscanf — Read Data" on page 530
"<stdio.h>" on page 774

fseek — Reposition File Position

Format

#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);

Description

Language Level: ANSI, SAA, POSIX, XPG4

fseek changes the current file position associated with *stream* to a new location within the file. The next operation on the *stream* takes place at the new location. On a *stream* open for update, the next operation can be either a reading or a writing operation.

The *origin* must be one of the following constants defined in <stdio.h>:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

For a binary stream, you can also change the position beyond the end of the file. An attempt to position before the beginning of the file causes an error. If successful, fseek clears the end-of-file indicator, even when *origin* is SEEK_END, and undoes the effect of any preceding ungetc function on the same stream.

Note: For streams opened in text mode, fseek has limited use because some system translations (such as those between carriage-return-line-feed and new line) can produce unexpected results. The only fseek operations that can be relied upon to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values or seeking from the beginning of the file with an offset value returned from a call to ftell. See the chapter "Performing I/O Operations" in the *Programming Guide* for more information.

Return Value

fseek returns 0 if it successfully moves the pointer. A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

This example opens a file myfile.dat for reading. After performing input operations (not shown), fseek moves the file pointer to the beginning of the file.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE  *stream;
   int  result;

    stream = fopen("myfile.dat", "r");

    result = fseek(stream, 0L, SEEK_SET);     /* moves the pointer to the
                                                 beginning of the file                */

    if (result)                               /* fail to move the pointer to the     */
        return EXIT_FAILURE;                      /* beginning of the file           */
   fclose(stream);
   return  0;
}
```

"fgetpos — Get File Position" on page 203
"fsetpos — Set File Position" on page 249
"ftell — Get Current Position" on page 257
"rewind — Adjust Current File Position" on page 465
"ungetc — Push Character onto Input Stream" on page 681
"<stdio.h>" on page 774

fsetpos — Set File Position

Format

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

Language Level: ANSI, SAA, XPG4

fsetpos moves any file position associated with *stream* to a new location within the file according to the value pointed to by *pos*. The value of *pos* was obtained by a previous call to the fgetpos library function.

If successful, fsetpos clears the end-of-file indicator, and undoes the effect of any previous ungetc function on the same stream.

After the fsetpos call, the next operation on a stream in update mode may be input or output.

Return Value

If fsetpos successfully changes the current position of the file, it returns 0. A nonzero return value indicates an error.

This example opens a file myfile.dat for reading. After performing input operations (not shown), fsetpos moves the file pointer to the beginning of the file and rereads the first byte.

```
#include <stdio.h>

FILE *stream;

int main(void)
{
   int  retcode;
   fpos_t  pos,pos1,pos2,pos3;
    char ptr[20];                  /* existing file 'myfile.dat' has 20 byte records    */
```

```
    /* Open file, get position of file pointer, and read first record              */

   stream = fopen("myfile.dat", "rb");
  fgetpos(stream,  &pos);
   pos1 = pos;
   if (!fread(ptr, sizeof(ptr), 1, stream))
       perror("fread  error");

    /* Perform a number of read operations - the value of 'pos' changes           */
    /* Re-set pointer to start of file and re-read first record                   */

  fsetpos(stream,  &pos1);
   if (!fread(ptr, sizeof(ptr), 1, stream))
       perror("fread  error");
  fclose(stream);
  return  0;
}
```

"fgetpos — Get File Position" on page 203
"fseek — Reposition File Position" on page 247
"ftell — Get Current Position" on page 257
"rewind — Adjust Current File Position" on page 465
"<stdio.h>" on page 774

_fsin — Calculate Sine

Format

```
#include <builtin.h>
double _fsin( double x);
```

Description

Language Level: Extension

_fsin calculates the sine of *x*, where *x* is expressed in radians. The value of *x* must be less than 2**63. This function causes the compiler to emit the appropriate 80387 instruction for the calculation of sine.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fsin.
You cannot parenthesize a _fsin function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

This function returns the sine of a variable *x* expressed in radians.

This example calculates *y* as the sine of π/*2*.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  pi;

    pi = 3.1415926535;
    printf("The sine of %lf is %lf.\n", pi/2.0, _fsin(pi/2.0));
   return  0;

   /*****************************************************************************
        The output should be :

        The sine of 1.570796 is 1.000000.
   *****************************************************************************/
}
```

"_fasin — Calculate Arcsine" on page 185
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"<builtin.h>" on page 761

_fsincos — Calculate Sine and Cosine

Format

```
#include <builtin.h>
double _fsincos(double x, double *y);
```

Description

Language Level: Extension

_fsincos calculates the sine of *x*, and stores the cosine in **y*. This operation is faster than separately calculating the sine and cosine. Use _fsincos instead of _fcossin when you will be using the sine first, and then the cosine.

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fsincos.
You cannot parenthesize a _fsincos function call because parentheses specify a call to the backing code for the function.

Return Value

This function returns the sine of *x*.

This example calculates the sine of x and stores it in z, and stores the cosine of x in y.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;

   printf("Enter  x:\n");
   scanf("%lf",  &x);

    z = _fsincos(x, &y);
    printf("The sine of %lf is %lf.\n", x, z);
    printf("The cosine of %lf is %lf.\n", x, y);
   return  0;

   /*****************************************************************************
        Assuming you enter : 1.0

        The output should be :

        The sine of 1.000000 is 0.841471.
        The cosine of 1.000000 is 0.540302.
   *****************************************************************************/
}
```

"acos — Calculate Arccosine" on page 43
"asin — Calculate Arcsine" on page 49
"cos — Calculate Cosine" on page 96
"cosh — Calculate Hyperbolic Cosine" on page 97
"_facos — Calculate Arccosine" on page 183

"_fasin — Calculate Arcsine" on page 185
"_fcos — Calculate Cosine" on page 189
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsin — Calculate Sine" on page 251
"sin — Calculate Sine" on page 514
"sinh — Calculate Hyperbolic Sine" on page 515
"<builtin.h>" on page 761

_fsqrt — Calculate Square Root

Format

```
#include <builtin.h>
double _fsqrt(double x);
```

Description

Language Level: Extension

_fsqrt computes the square root of *x* using the FSQRT 80387 instruction. Note that this function does not set errno as the sqrt function does. If you pass a negative value to this function, an exception is generated.

Note: _fsqrt is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _fsqrt.
You cannot parenthesize a call to _fsqrt. (Parentheses specify a call to the function's backing code, and _fsqrt has none.)

Return Value

_fsqrt returns the value of the square root of *x*.

This example calculates the square root of 4.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  x;

    x = 4.0;
    printf("The square root of %lf is %lf.\n", x, _fsqrt(x));
   return  0;

   /*****************************************************************************
        The output should be :

        The square root of 4.000000 is 2.000000.
   *****************************************************************************/
}
```

"sqrt — Calculate Square Root" on page 527
"<math.h>" on page 770

fstat — Information about Open File

Format

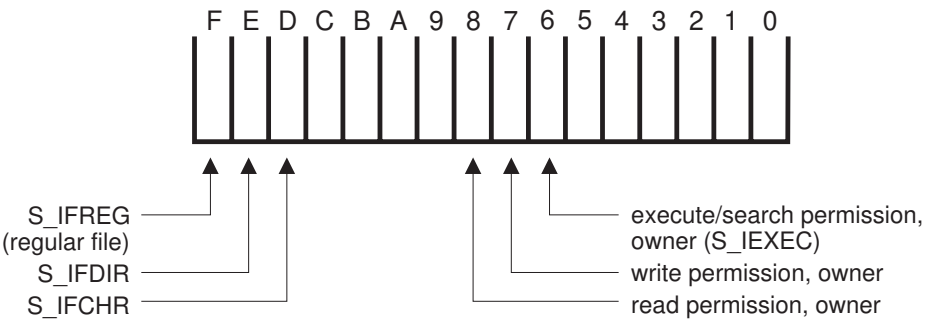
```
#include <sys\stat.h>
#include <sys\types.h>
int fstat(int handle, struct stat *buffer);
```

Description

Language Level: XPG4, Extension

fstat obtains information about the open file associated with the given *handle* and stores it in the structure to which *buffer* points. The <sys\stat.h> include file defines the stat structure. The stat structure contains the following fields:

Field	Value
st_mode	Bit mask for file mode information. fstat sets the S_IFCHR bit if *handle* refers to a device. The S_IFDIR bit is set if *pathname* specifies a directory. It sets the S_IFREG bit if *handle* refers to an ordinary file. It sets user read/write bits according to the permission mode of the file. The other bits have undefined values.



st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.
st_size	Size of the file in bytes.
st_atime	Time of last access of file.
st_mtime	Time of last modification of file.
st_ctime	Time of file creation.

There are three additional fields in the stat structure for portability to other operating systems; they have no meaning under Windows.

Notes:

1. If the given *handle* refers to a device, the size and time fields in the stat structure are not meaningful.
2. In earlier releases of VisualAge C++, fstat began with an underscore (_fstat). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _fstat to fstat for you.

Return Value

If it obtains the file status information, fstat returns 0. A return value of -1 indicates an error, and fstat sets **errno** to EBADF, showing an incorrect file handle.

This example uses fstat to report the size of a file named data.

```
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>

int main(void)
{
    struct stat buf;
   FILE  *fp;
   int  fh,result;
    char *buffer = "A line to output";

    fp = fopen("data", "w+");
    fprintf(fp, "%s", buffer);
   fflush(fp);
   fclose(fp);
    fp = fopen("data", "r");
    if (0 == fstat(fileno(fp), &buf)) {
        printf("file size is %ld\n", buf.st_size);
        printf("time modified is %s\n", ctime(&buf.st_atime));
   }
   else
        printf("Bad file handle\n");
   fclose(fp);
   return  0;

   /*************************************************************************
        The output should be similar to:

        file size is 16
        time modified is Thu May 16 16:08:14 1995
   *************************************************************************/
}
```

"stat — Get Information about File or Directory" on page 532
"<sys\stat.h>" on page 778
"<sys\types.h>" on page 778

ftell — Get Current Position

Format

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ftell finds the current position of the file associated with *stream*. For a fixed-length binary file, the value returned by ftell is an offset relative to the beginning of the *stream*.

Note: For buffered text streams, ftell returns an incorrect file position if the file contains new-line characters instead of carriage-return line-feed combinations. Your file would only contain new-line characters if you previously used it as a binary stream. To avoid this problem, either continue to process the file as a binary stream, or use unbuffered I/O operations.

Return Value

ftell returns the current file position. On error, ftell returns -1L and sets errno to a nonzero value.

This example opens the file myfile.dat for reading. It reads enough characters to fill half of the buffer and prints out the position in the stream and the buffer.

ftell

```
#include <stdio.h>

#define  NUM_ALPHA        26
#define  NUM_CHAR         6

int main(void)
{
   FILE  *stream;
   int  i;
   char  ch;
   char  buffer[NUM_ALPHA];
   long  position;

    if ((stream = fopen("myfile.dat", "r")) != NULL) {

      /* read into buffer                                                    */

       for (i = 0; (i < NUM_ALPHA/2) && ((buffer[i] = fgetc(stream)) != EOF);
            ++i)
          if (i == NUM_CHAR-1) { /* We want to be able to position the
                                    file pointer to the character in
                                    position  NUM_CHAR                      */

              position = ftell(stream);
              printf("Current position of the file is stored.\n");
          }
       buffer[i] = '\0';
       fseek(stream, position, SEEK_SET);
       ch = fgetc(stream);          /* get the character at position NUM_CHAR  */
      fclose(stream);
   }
   else
        perror("Error opening myfile.dat");
   return  0;
}
```

"fseek — Reposition File Position" on page 247
"fgetpos — Get File Position" on page 203
"fopen — Open Files" on page 218
"fsetpos — Set File Position" on page 249
"<stdio.h>" on page 774

_ftime — Store Current Time

Format

```
#include <sys\timeb.h>
#include <sys\types.h>
void _ftime(struct timeb *timeptr);
```

Description

Language Level: Extension

_ftime gets the current time and stores it in the structure to which *timeptr* points. The <sys\timeb.h> include file contains the definition of the timeb structure. It contains four fields:

time The time in seconds since 00:00:00 Coordinated Universal Time, January 1, 1970.

millitm The fraction of a second, in milliseconds.

timezone The difference in minutes between Coordinated Universal Time and local time, going from east to west. _ftime sets the value of timezone from the value of the global variable _timezone.

dstflag Nonzero if daylight saving time is currently in effect for the local time zone. For an explanation of how daylight saving time is determined, see "tzset — Assign Values to Locale Information" on page 634.

Return Value

There is no return value.

This example polls the system clock, converts the current time to a character string, prints the string, and saves the time data in the structure timebuffer.

```
#include <sys\types.h>
#include <sys\timeb.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct timeb timebuffer;

   _ftime(&timebuffer);
    printf("the time is %s\n", ctime(&(timebuffer.time)));
   return  0;

   /*************************************************************************
        The output should be similar to:

        the time is Thu May 16 16:08:17 1995
   *************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289

_ftime

"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"time — Determine Current Time" on page 625
"tzset — Assign Values to Locale Information" on page 634
"<sys\timeb.h>" on page 778
"<sys\types.h>" on page 778

_fullpath — Get Full Path Name of Partial Path

Format

```
#include <stdlib.h>
char *_fullpath(char *pathbuf, char *partialpath, size_t n);
```

Description

Language Level: Extension

_fullpath gets the full path name of the given partial path name *partialpath*, and stores it in *pathbuf*. The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If *pathbuf* is NULL, _fullpath uses malloc to allocate a buffer of size _MAX_PATH bytes to store the path name.

Return Value

_fullpath returns a pointer to *pathbuf*. If an error occurs, _fullpath returns NULL and sets **errno** to one of the following values:

Value	Meaning
ENOMEM	Unable to allocate storage for the buffer.
ERANGE	The path name is longer than *n* characters.
EOS2ERR	A system error occurred. Check _doserrno for the specific Windows code.

This example uses _fullpath to get and store the full path name of the current directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
   char  *retBuffer;

    retBuffer = _fullpath(NULL, ".", 0);
    if (NULL == retBuffer) {
        printf("An error has occurred, errno is set to %d.\n", errno);
   }
   else
        printf("Full path name of current directory is %s.\n", retBuffer);
   return  0;

   /*****************************************************************************
        The output should be similar to:

           Full path name of current directory is D:\BIN.
   *****************************************************************************/
}
```

"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276

_fullpath

"_makepath — Create Path" on page 374
"malloc — Reserve Storage Block" on page 376
"_splitpath — Decompose Path Name" on page 523
"<stdlib.h>" on page 775

fwrite — Write Items

Format

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
                        FILE  *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

fwrite writes up to *count* items, each of *size* bytes in length, from *buffer* to the output *stream*.

Return Value

fwrite returns the number of full items successfully written, which can be fewer than *count* if an error occurs.

This example writes NUM long integers to a stream in binary format.

```
#include <stdio.h>

#define   NUM                 100

int main(void)
{
    FILE  *stream;
    long  list[NUM];
    int  numwritten;

     stream = fopen("myfile.dat", "w+b");

                    /* assign values to list[]                                    */

     numwritten = fwrite(list, sizeof(long), NUM, stream);
     printf("Number of items successfully written : %d\n", numwritten);
    return  0;

   /*****************************************************************************
        The output should be:

        Number of items successfully written : 100
   *****************************************************************************/
}
```

"fopen — Open Files" on page 218
"fread — Read Items" on page 236
"read — Read Into Buffer" on page 450
"write — Writes from Buffer to File" on page 759
"<stdio.h>" on page 774

_fyl2x — Calculate y * log2(x)

Format

#include <builtin.h>
double _fyl2x(double *x*, double *y*);

Description

Language Level: Extension

_fyl2x computes the base-2 logarithm of *x* and multiplies it by *y* (*y* * log2(*x*)). The variable *x* cannot be negative. This instruction is designed with built-in multiplication to optimize the calculation of logarithms with arbitrary positive base: log *b* (*x*) = (log2(*b*)**-1) * log2(*x*)

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _fyl2x.
You cannot parenthesize a _fyl2x function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

_fyl2x returns the result of the formula *y* * log2(*x*).

This example calculates (*y* * log2(*x*)), after prompting the user for values of x and y.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
    double x, y;

    printf("Enter a value for x:\n");
   scanf("%lf",  &x);
    printf("Enter a value for y:\n");
   scanf("%lf",  &y);
    printf("%lf * log2(%lf) is %lf.\n", y, x, _fyl2x(x, y));
   return  0;

   /*****************************************************************************
        Assuming you enter 4.0 for x and 3.0 for y.

        The output should be :

        3.000000 * log2(4.000000) is 6.000000.
   *****************************************************************************/
}
```

"_fyl2xp1 — Calculate y * log2(x + 1)" on page 265
"_f2xm1 — Calculate (2**x) – 1" on page 266
"<builtin.h>" on page 761

_fyl2xp1 — Calculate y * log2(x + 1)

Format

#include <builtin.h>
double _fyl2xp1(double *x*, double *y*);

Description

Language Level: Extension

_fyl2xp1 computes the base-2 logarithm of *x*+1 and multiplies it by *y* (*y* * log2(*x*+1)). The variable *x* must be in the range -(1-(sqrt(2)/2)) to (sqrt(2) - 1). _fyl2xp1 provides improved accuracy over _fyl2x when logarithms of numbers very close to 1 are computed. When *x* is small, you can retain more significant digits by providing *x* to the _fyl2xp1 function than by providing *x*+1 as an argument to the _fyl2x function.

Because it is a built-in function and has no backing code in the library:

- You cannot take the address of _fyl2xp1.
- You cannot parenthesize a _fyl2xp1 function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

_fyl2xp1 returns the result of the formula *y* * log2(*x*+1).

This example calculates (*y* * log2(*x* + 1)), after prompting the user for values of x and y.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
    double x, y;

    printf("Enter a value for x:\n");
   scanf("%lf",  &x);
    printf("Enter a value for y:\n");
   scanf("%lf",  &y);
    printf("%lf * log2(%lf + 1) is %lf.\n", y, x, _fyl2xp1(x, y));
   return  0;

   /*************************************************************************
        Assuming you enter 0.001 for x and 2.0 for y.

        The output should be :

        2.000000 * log2(0.001000 + 1) is 0.002884.
   *************************************************************************/
}
```

"_fyl2x — Calculate y * log2(x)" on page 264
"_f2xm1 — Calculate (2**x) – 1" on page 266
"<builtin.h>" on page 761

_f2xm1 — Calculate (2**x) – 1

Format

```
#include <builtin.h>
double _f2xm1(double x);
```

Description

Language Level: Extension

_f2xm1 calculates 2 raised to the power of *x*, minus 1 ((2**x)-1). The variable *x* must be in the range -1 to 1. This instruction is designed to produce very accurate results when *x* is close to 0. Large errors may occur for operands with magnitudes close to 1. You can exponentiate values other than 2 by using the formula *x****y* = 2**(*y* * log2(*x*)).

Because it is a built-in function and has no backing code in the library:

You cannot take the address of _f2xm1.
You cannot parenthesize a _f2xm1 function call, because parentheses specify a call to the backing code for the function in the library.

Return Value

This function returns the value of the formula (2**x)-1.

This example calculates (2**x)-1 after prompting the user for a value for x.

```
#include <builtin.h>
#include <stdio.h>

int main(void)
{
   double  x;

    printf("Enter a value for x:\n");
   scanf("%lf",  &x);
    printf("(2**(%lf)) - 1 is %lf.\n", x, _f2xm1(x));
   return  0;

   /*****************************************************************************
        Assuming you enter : 0.001

        The output should be :

        (2**(0.001000)) - 1 is 0.000693.
   *****************************************************************************/
}
```

"_fyl2x — Calculate y * log2(x)" on page 264
"_fyl2xp1 — Calculate y * log2(x + 1)" on page 265
"<builtin.h>" on page 761

gamma — Gamma Function

Format

```
#include <math.h> /* SAA extension to ANSI */
double gamma(double x);
```

Description

Language Level: SAA

gamma computes the natural logarithm of the absolute value of G(x) (ln(|G(x)|)), where

$$G(x) = \int_0^\infty e^{-t} \times t^{x-1}\, dt$$

Portability Note: According to the SAA standard, *x* must be a positive real value. Under the VisualAge for C++ compiler, *x* can also be a negative integer.

Return Value

gamma returns the value of ln(|G(x)|). If *x* is a negative value, errno is set to EDOM. If the result causes an overflow, gamma returns HUGE_VAL and sets errno to ERANGE.

This example uses gamma to calculate ln(|G(x)|), where x = 42.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x = 42,g_at_x;

    g_at_x = exp(gamma(x));                          /* g_at_x = 3.345253e+49        */
    printf("The value of G(%4.2f) is %7.2e\n", x, g_at_x);
   return  0;

  /*****************************************************************************
       The output should be:

       The value of G(42.00) is 3.35e+49
  *****************************************************************************/
}
```

"Bessel Functions — Solve Differential Equations" on page 69
"erf – erfc — Calculate Error Functions" on page 173
"<math.h>" on page 770

_gcvt — Convert Floating-Point to String

Format

#include <stdlib.h>
char *_gcvt(double *value*, int *ndec*, char **buffer*);

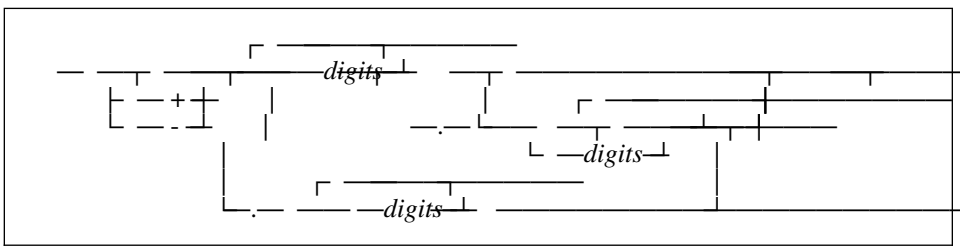
Description

Language Level: Extension

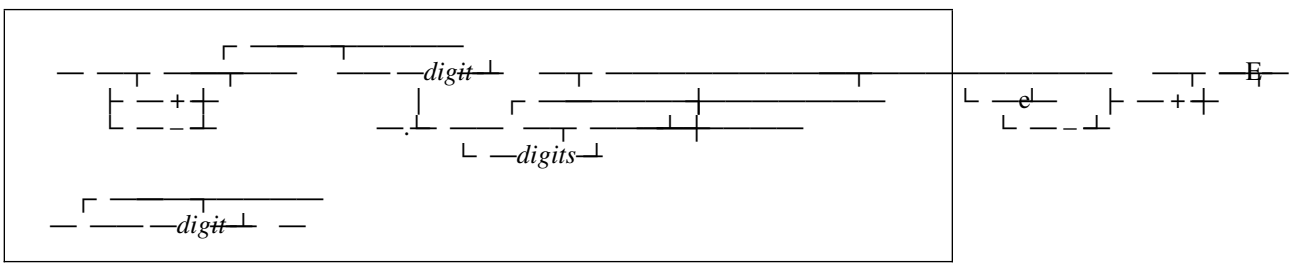
_gcvt converts a floating-point *value* to a character string pointed to by *buffer*. The *buffer* should be large enough to hold the converted value and a null character (\0) that _gcvt automatically adds to the end of the string. There is no provision for overflow.

_gcvt tries to produce *ndec* significant digits in FORTRAN F format. Failing that, it produces *ndec* significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion if they are insignificant.

A FORTRAN F number has the following format:



A FORTRAN E number has the following format:



_gcvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively. For more information on NaN and infinity values, see "Infinity and NaN Support" on page 27.

Warning: For each thread, _ecvt, _fcvt and _gcvt use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Return Value

_gcvt returns a pointer to the string of digits. If it cannot allocate memory to perform the conversion, _gcvt returns an empty string and sets **errno** to ENOMEM.

This example converts the value −3.1415e3 to a character string and places it in the character array buffer1. It then converts the macro value _INF to a character string and places it in buffer2.

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>                      /* for the definition of _INF                  */

int main(void)
{
   char  buffer1[10],buffer2[10];

    _gcvt(-3.1415e3, 7, buffer1);
    printf("The first result is %s \n", buffer1);
    _gcvt(_INF, 5, buffer2);
    printf("The second result is %s \n", buffer2);
   return  0;

   /*****************************************************************************
        The output should be:

        The first result is -3141.5
        The second result is INFINITY
   *****************************************************************************/
}
```

"_ecvt — Convert Floating-Point to Character" on page 166
"_fcvt — Convert Floating-Point to String" on page 192
"Infinity and NaN Support" on page 27
"<stdlib.h>" on page 775

getc – getchar — Read a Character

Format

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

getc reads a single character from the current *stream* position and advances the *stream* position to the next character. getchar is identical to getc(stdin).

getc is equivalent to fgetc except that, if it is implemented as a macro, getc can evaluate *stream* more than once. Therefore, the *stream* argument to getc should not be an expression with side effects.

Return Value

getc and getchar return the character read. A return value of EOF indicates an error or end-of-file condition. Use ferror or feof to determine whether an error or an end-of-file condition occurred.

This example gets a line of input from the **stdin** stream. You can also use getc(stdin) instead of getchar() in the for statement to get a line of input from **stdin**.

```
#include <stdio.h>
#define LINE 80

int main(void) {

   char  buffer[LINE+1];
   int  i;
   int  ch;
    printf( "Please enter string\n" );

    /* Keep reading until either:
         1. the length of LINE is exceeded    or
         2. the input character is EOF    or
         3. the input character is a newline character
     */
    for ( i = 0; ( i < LINE ) && (( ch = getchar()) != EOF) &&
                        ( ch !='\n' ); ++i )
       buffer[i] = ch;
    buffer[i] = '\0';    /* a string should always end with '\0' ! */
    printf( "The string is %s\n", buffer );
   return  0;
```

```
/*****************************************************************
Output should be:

Please enter string
hello world
The string is hello world
*****************************************************************/
}
```

"fgetc — Read a Character" on page 200
"_fgetchar — Read Single Character from **stdin**" on page 202
"_getch - _getche — Read Character from Keyboard" on page 272
"putc – putchar — Write a Character" on page 436
"ungetc — Push Character onto Input Stream" on page 681
"gets — Read a Line" on page 281
"<stdio.h>" on page 774

_getch - _getche — Read Character from Keyboard

Format

```
#include <conio.h>
int _getch(void);
int _getche(void);
```

Description

Language Level: Extension

_getch reads a single character from the keyboard, without echoing. _getche reads a single character from the keyboard and displays the character read. Neither function can be used to read Ctrl-Break.

You can use _kbhit to test if a keystroke is waiting in the buffer. If you call _getch or _getche without first calling _kbhit, the program waits for a key to be pressed.

Return Value

_getch and _getche return the character read. To read a function key or cursor-moving key, you must call _getch and _getche twice; the first call returns 0 or 0xE0, and the second call returns the particular extended key code.

This example gets characters from the keyboard until it finds the character 'x'.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
    int  ch;
     printf("\nType in some letters.\n");
     printf("If you type in an 'x', the program ends.\n");
    for(;;)  {
          ch = _getch();
          if ('x' == ch) {
               _ungetch(ch);
               break;
          }
          _putch(ch);
    }
     ch = _getch();
    printf("\n");
     printf("\nThe last character was '%c'.", ch);
    return  0;
```

```
    /*************************************************************
        Here is the output from a sample run:

        Type in some letters.
        If you type in an 'x', the program ends.
        One Two Three Four Five Si
        The last character was 'x'.
    *************************************************************/
}
```

"_cgets — Read String of Characters from Keyboard" on page 78
"fgetc — Read a Character" on page 200
"getc – getchar — Read a Character" on page 270
"_kbhit — Test for Keystroke" on page 337
"_putch — Write Character to Screen" on page 438
"_ungetch — Push Character Back to Keyboard" on page 683
"<conio.h>" on page 762

_getcwd — Get Path Name of Current Directory

Format

#include <direct.h>
char *_getcwd(char **pathbuf*, int *n*);

Description

Language Level: Extension

_getcwd gets the full path name of the current working directory and stores it in the buffer pointed to by *pathbuf*. The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If the *pathbuf* argument is NULL, _getcwd uses malloc to reserve a buffer of at least *n* bytes to store the path name. If the current working directory string is more than *n* bytes, a large enough buffer will be allocated to contain the string. You can later free this buffer using the _getcwd return value as the argument to free.

Return Value

_getcwd returns *pathbuf*. If an error occurs, _getcwd returns NULL and sets **errno** to one of the following values:

Value	Meaning
ENOMEM	Not enough storage space available to reserve *n* bytes (when *pathbuf* is NULL).
ERANGE	The path name is longer than *n* characters.
EOS2ERR	A Windows system call failed. Use _doserrno to obtain more information about the return code.

This example stores the name of the current working directory (up to _MAX_PATH characters) in a buffer. The value of _MAX_PATH is defined in <stdlib.h>.

```
#include <direct.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  buffer[_MAX_PATH];

    if (NULL == getcwd(buffer, _MAX_PATH))
       perror("getcwd  error");
    printf("The current directory is %s\n", buffer);
   return  0;

   /*****************************************************************************
        The output should be similar to:

        The current directory is E:\MIG_XMPS
   *****************************************************************************/
}
```

"chdir — Change Current Working Directory" on page 80
"_fullpath — Get Full Path Name of Partial Path" on page 261
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"_getdrive — Get Current Working Drive" on page 278
"malloc — Reserve Storage Block" on page 376
"<direct.h>" on page 762

_getdcwd — Get Full Path Name of Current Directory

Format

#include <direct.h>
char *_getdcwd(int *drive*, char **pathbuf*, int *n*);

Description

Language Level: Extension

_getdcwd gets the full path name for the current directory of the specified *drive*, and stores it in the location pointed to by *pathbuf*. The *drive* argument is an integer value representing the drive (A: is 1, B: is 2, and so on).

The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If the *pathbuf* argument is NULL, _getdcwd uses malloc to reserve a buffer of at least *n* bytes to store the path name. If the current working directory string is more than *n* bytes, a large enough buffer will be allocated to contain the string. You can later free this buffer using the _getdcwd return value as the argument to free.

The alternative to this function is the GetCurrentDirectory API call.

Return Value

_getdcwd returns *pathbuf*. If an error occurs, _getdcwd returns NULL and sets errno to one of the following values:

Value	Meaning
ENOMEM	Not enough storage space available to reserve *n* bytes (when *pathbuf* is NULL).
ERANGE	The path name is longer than *n* characters.
EOS2ERR	A Windows system call failed. Use _doserrno to obtain more information about the return code.

This example uses _getdcwd to obtain the current working directory of drive C.

```
#include <stdio.h>
#include <direct.h>
#include <errno.h>

int main(void)
{
   char  *retBuffer;

    retBuffer = _getdcwd(3, NULL, 0);
    if (NULL == retBuffer)
        printf("An error has occurred, errno is set to %d.\n", errno);
   else
        printf("%s is the current working directory in drive C.\n", retBuffer);
   return  0;

   /*************************************************************************
        The output should be similar to:

        C:\ is the current working directory in drive C.
   *************************************************************************/
}
```

"chdir — Change Current Working Directory" on page 80
"_chdrive — Change Current Working Drive" on page 82
"_fullpath — Get Full Path Name of Partial Path" on page 261
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdrive — Get Current Working Drive" on page 278
"malloc — Reserve Storage Block" on page 376
"<direct.h>" on page 762

_getdrive — Get Current Working Drive

Format

```
#include <direct.h>
int _getdrive(void);
```

Description

Language Level: Extension

_getdrive gets the drive number for the current working drive.

An alternative to this function is the GetCurrrentDirectory API call.

Return Value

_getdrive returns an integer corresponding to alphabetical position of the letter representing the current working drive. For example, A: is 1, B: is 2, J: is 10, and so on.

This example gets and prints the current working drive number.

```
#include <stdio.h>
#include <direct.h>

int main(void)
{
    printf("Current working drive is %d.\n", _getdrive());
   return  0;

   /*****************************************************************************
        The output should be similar to:

        Current working drive is 5.
   *****************************************************************************/
}
```

"chdir — Change Current Working Directory" on page 80
"_chdrive — Change Current Working Drive" on page 82
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"<direct.h>" on page 762

getenv — Search for Environment Variables

Format

```
#include <stdlib.h>
char *getenv(const char *varname);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

getenv searches the list of environment variables for an entry corresponding to *varname*.

Return Value

getenv returns a pointer to the environment table entry containing the current string value of *varname*. The return value is NULL if the given variable is not currently defined or if the system does not support environment variables.

You should copy the string that is returned because it may be written over by a subsequent call to getenv.

In this example, *pathvar* points to the value of the PATH environment variable.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  *pathvar;

    pathvar = getenv("PATH");
    if (NULL == pathvar)
        printf("Environment variable PATH is not defined.\n");
   else
        printf("Path set by environment variable PATH is successfully stored.\n");
   return  0;

   /*****************************************************************************
        The output should be:

        Path set by environment variable PATH is successfully stored.
   *****************************************************************************/
}
```

"putenv — Modify Environment Variables" on page 439
"<stdlib.h>" on page 775

getpid — Get Process Identifier

Format

```
#include <process.h>
int getpid(void);
```

Description

Language Level: XPG4, Extension

getpid gets the process identifier that uniquely identifies the calling process.

Note: In earlier releases of VisualAge C++, getpid began with an underscore (_getpid). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _getpid to getpid for you.

Return Value

getpid function gets the process identifier as an integer value. There is no error return value.

This example prints the process identifier:

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Process identifier is %05d\n", getpid());
   return  0;

   /*************************************************************************
       The output should be similar to:

       Process identifier is 00242
   *************************************************************************/
}
```

"_cwait — Wait for Child Process" on page 116
"execl - _execvpe — Load and Run Child Process" on page 175
"_spawnl - _spawnvpe — Start and Run Child Processes" on page 519
"<process.h>" on page 772

gets — Read a Line

Format

```
#include <stdio.h>
char *gets(char *buffer);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

gets reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first new-line character (\n) or EOF. gets then replaces the new-line character, if read, with a null character (\0) before returning the line.

Return Value

If successful, gets returns its argument. A NULL pointer return value indicates an error or an end-of-file condition with no characters read. Use ferror or feof to determine which of these conditions occurred. If there is an error, the value stored in *buffer* is undefined. If an end-of-file condition occurs, *buffer* is not changed.

This example gets a line of input from **stdin**.

```
#include <stdio.h>

#define   MAX_LINE          100

int main(void)
{
   char  line[MAX_LINE];
   char  *result;

    if ((result = gets(line)) != NULL) {
       if  (ferror(stdin))
            perror("Error");
        printf("Input line : %s\n", result);
   }
   return  0;

   /*****************************************************************************
        For the following input:
        This is a test for function gets.

        The output should be:
        Input line : This a test for function gets.
   *****************************************************************************/
}
```

"_cgets — Read String of Characters from Keyboard" on page 78
"fgets — Read a String" on page 205
"feof — Test End-of-File Indicator" on page 197
"ferror — Test for Read/Write Errors" on page 198
"fputs — Write String" on page 230
"getc – getchar — Read a Character" on page 270
"puts — Write a String" on page 441

gets

"<stdio.h>" on page 774

getsyntx — Return LC_SYNTAX Characters

Format

```
#include <variant.h>
struct variant *getsyntx(void);
```

Description

Language Level: Extension

getsyntx determines the encoding of the special characters defined in the LC_SYNTAX category of the current locale, and stores the encoding values in the structure of type struct variant. For details of the structure type, see "<variant.h>" on page 780.

Your program cannot modify the returned structure. The structure can be overwritten by a call to setlocale with the argument LC_ALL or LC_SYNTAX.

Return Value

getsyntx returns the pointer to the structure containing the values of the special characters. If the information about the special characters is not available in the current locale, getsyntx returns a null pointer.

This example uses getsyntx to show the value of various special characters.

```
#include <stdio.h>
#include <stdlib.h>
#include <variant.h>
#include <locale.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "en_us.ibm-850"        /* OS/2 name             */
#else
    #define LOCNAME "fr_fr.ibm-1252"       /* Windows name          */
#endif

int main(void)
{
    struct variant *var;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
    }
```

```
 var = getsyntx();
 printf("codeset                : %s\n", var->codeset                );
 printf("backslash              : %c\n", var->backslash              );
printf("right_bracket     :  %c\n",  var->right_bracket     );
 printf("left_bracket        : %c\n", var->left_bracket        );
 printf("right_brace          : %c\n", var->right_brace          );
 printf("left_brace           : %c\n", var->left_brace           );
 printf("circumflex            : %c\n", var->circumflex           );
 printf("tilde                  : %c\n", var->tilde                  );
 printf("exclamation_mark: %c\n", var->exclamation_mark);
 printf("number_sign          : %c\n", var->number_sign          );
printf("vertical_line     :  %c\n",  var->vertical_line     );
 printf("dollar_sign          : %c\n", var->dollar_sign          );
printf("commercial_at     :  %c\n",  var->commercial_at     );
 printf("grave_accent        : %c\n", var->grave_accent        );
return  0;

/*****************************************************************************
     The OS/2 output should be similar to :

    codeset                :  IBM-850
    backslash              :  \
    right_bracket      :  ]
    left_bracket       :  [
    right_brace         :  }
    left_brace          :  {
    circumflex          :  ¬
    tilde                  :
    exclamation_mark:  !
    number_sign          :  #
    vertical_line      :  |
    dollar_sign          :  $
    commercial_at      :  @
    grave_accent       :
*****************************************************************************/
}
```

"setlocale — Set Locale" on page 499
"<locale.h>" on page 766
"<variant.h>" on page 780

getwc — Read Wide Character from Stream

Format

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

Language Level: ANSI 93, XPG4

getwc reads the next multibyte character from *stream*, converts it to a wide character, and advances the associated file position indicator for *stream*.

getwc is equivalent to fgetwc except that, if it is implemented as a macro, it can evaluate *stream* more than once. Therefore, the argument should never be an expression with side effects.

The behavior of getwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

Using non-wide-character functions with getwc on the same stream results in undefined behavior.

After calling getwc, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling getwc.

Return Value

getwc returns the next wide character from the input stream, or WEOF. If an error occurs, getwc sets the error indicator. If getwc encounters the end-of-file, it sets the EOF indicator. If an encoding error occurs during conversion of the multibyte character, getwc sets **errno** to EILSEQ.

Use ferror or feof to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

This example opens a file and uses getwc to read wide characters from the file.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   FILE       *stream;
   wint_t    wc;

    if (NULL == (stream = fopen("getwc.dat", "r"))) {
```

```
        printf("Unable to open: \"getwc.dat\".\n");
        exit(1);
    }

    errno = 0;
    while (WEOF !=(wc = getwc(stream)))
        printf("wc = %lc\n", wc);

    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return  0;

    /*************************************************************************
        Assuming the file getwc.dat contains:

        Hello  world!

        The output should be:

        wc = H
        wc = e
        wc = l
        wc = l
        wc = o
        :
    *************************************************************************/
}
```

"fgetwc — Read Wide Character from Stream" on page 207
"getwchar — Get Wide Character from stdin" on page 287
"getc – getchar — Read a Character" on page 270
"_getch - _getche — Read Character from Keyboard" on page 272
"putwc — Write Wide Character" on page 442
"<stdio.h>" on page 774
"<wchar.h>" on page 780

getwchar — Get Wide Character from stdin

Format

```
#include <wchar.h>
wint_t getwchar(void);
```

Description

Language Level: ANSI, SAA, XPG4

getwchar reads the next multibyte character from **stdin**, converts it to a wide character, and advances the associated file position indicator for **stdin**. A call to getwchar is equivalent to a call to getwc(stdin).

The behavior of getwchar is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

Using non-wide-character functions with getwchar on STDIN results in undefined behavior.

Return Value

getwchar returns the next wide character from **stdin** or WEOF. If getwchar encounters EOF, it sets the EOF indicator for the stream and returns WEOF. If a read error occurs, the error indicator for the stream is set and getwchar returns WEOF. If an encoding error occurs during the conversion of the multibyte character to a wide character, getwchar sets **errno** to EILSEQ and returns WEOF.

Use ferror or feof to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

This example uses getwchar to read wide characters from the keyboard, then prints the wide characters.

getwchar

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wint_t    wc;

     errno = 0;
     while (WEOF != (wc = getwchar()))
          printf("wc = %lc\n", wc);

     if (EILSEQ == errno) {
          printf("An invalid wide character was encountered.\n");
          exit(1);
     }
    return  0;

    /*********************************************************************************
          Assuming you enter: abcde Z                    (note:  Z is CNTRL-Z)

          The output should be:

          wc = a
          wc = b
          wc = c
          wc = d
          wc = e
    *********************************************************************************/
}
```

"fgetc — Read a Character" on page 200
"_fgetchar — Read Single Character from **stdin**" on page 202
"getc – getchar — Read a Character" on page 270
"_getch - _getche — Read Character from Keyboard" on page 272
"getwc — Read Wide Character from Stream" on page 285
"<wchar.h>" on page 780

gmtime — Convert Time

Format

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

The gmtime function breaks down the *time* value and stores it in a tm structure, defined in time.h. The structure pointed to by the return value reflects Coordinated Universal Time, not local time. The value *time* is usually obtained from a call to time.

The fields of the tm structure include:

tm_sec	Seconds (0-61)
tm_min	Minutes (0-59)
tm_hour	Hours (0-23)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Zero if Daylight Saving Time is not in effect; positive if Daylight Saving Time is in effect; negative if the information is not available.

Return Value

gmtime returns a pointer to the resulting tm structure. It returns NULL if Coordinated Universal Time is not available.

Notes:

1. The range (0-61) for tm_sec allows for as many as two leap seconds.
2. The gmtime and localtime functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
3. The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

This example uses gmtime to adjust a time_t representation to a Coordinated Universal Time character string, and then converts it to a printable string using asctime.

gmtime

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t  ltime;

    time(&ltime);
     printf("Coordinated Universal Time is %s\n", asctime(gmtime(&ltime)));
    return  0;

    /*****************************************************************************
         The output should be similar to:

         Coordinated Universal Time is Mon Sep 16 21:44 1995
    *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"time — Determine Current Time" on page 625
"<time.h>" on page 779

_heap_check — Validate Default Memory Heap

Format

```
#include <stdlib.h>       /* also in <malloc.h> */
void _heap_check(void);
```

Description

Language Level: Extension

_heap_check checks all memory blocks in the default heap that have been allocated or freed using the debug memory management functions (_debug_calloc, _debug_malloc, and so on). _heap_check checks that your program has not overwritten freed memory blocks or memory outside the bounds of allocated blocks.

When you call a debug memory management function (such as _debug_malloc), it calls _heap_check automatically. You can also call _heap_check explicitly. Place calls to _heap_check throughout your code, especially in areas that you suspect have memory problems.

Calling _heap_check frequently (explicitly or through the debug memory functions) can increase your program's memory requirements and decrease its execution speed. To reduce the overhead of heap-checking, you can use the CPP_HEAP_SKIP environment variable to control how often the functions check the heap. For example:

```
SET CPP_HEAP_SKIP=10
```

specifies that every tenth call to any debug memory function (regardless of the type of heap) checks the heap. Explicit calls to _heap_check are always performed. For more details on CPP_HEAP_SKIP, see "Skipping Heap Checks" in the *Programming Guide*.

To use _heap_check and the debug memory management functions, you must compile with the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

_heap_check always checks the default heap.

Return Value There is no return value.

This example allocates 5000 bytes of storage, and then attempts to write to storage that was not allocated. The call to _heap_check detects the error, generates several messages, and stops the program.

_heap_check

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *ptr;

    if (NULL == (ptr = (char*)malloc(5000))) {
        puts("Could not allocate memory block.");
       return  EXIT_FAILURE;
   }

    *(ptr-1) = 'a';                    /* overwrites storage that was not allocated       */
   _heap_check();
    puts("_heap_check did not detect that a memory block was overwritten.");
   return  0;

   /*************************************************************************
        The output should be similar to:

        Header information of object 0x00073890 was overwritten at 0x0007388c.
        The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
        This memory block was (re)allocated at line number 8 in _heap_check.c.
        Heap state was valid at line 8 of _heap_check.c.
        Memory error detected at line 14 of _heap_check.c.
   *************************************************************************/
}
```

Memory Management in the *Programming Guide*
Debugging Your Heaps in the *Programming Guide*
"Differentiating between Memory Management Functions" on page 23
"_debug_calloc — Allocate and Initialize Memory" on page 124
"_debug_free — Release Memory" on page 126
"_debug_heapmin — Release Unused Memory in the Default Heap" on page 128
"_debug_malloc — Allocate Memory" on page 130
"_debug_realloc — Reallocate Memory Block" on page 132
"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_heapchk — Validate Default Memory Heap" on page 293
"_uheap_check — Validate User Memory Heap" on page 662
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_heapchk — Validate Default Memory Heap

Format

```
#include <malloc.h>
int _heapchk(void);
```

Description

Language Level: Extension

_heapchk checks the default storage heap for minimal consistency by checking all allocated and freed objects on the heap.

A heap-specific version of this function, _uheapchk, is also available.

Note: Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.

Return Value

_heapchk returns one of the following values, defined in <malloc.h>:

_HEAPBADBEGIN	The heap specified is not valid. (Only occurs if you changed the default heap and then later closed or destroyed it.)
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

This example performs some memory allocations, then calls _heapchk to check the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

int main(void)
{
   char  *ptr;
   int    rc;

    if (NULL == (ptr = (char*)malloc(10))) {
        puts("Could not allocate memory block.");
       exit(EXIT_FAILURE);
   }
```

```
    *(ptr - 1) = 'x';           /* overwrites storage that was not allocated */

    if (_HEAPOK != (rc = _heapchk())) {
        switch(rc)  {
            case  _HEAPEMPTY:
                  puts("The heap has not been initialized.");
                 break;
            case  _HEAPBADNODE:
                  puts("A memory node is corrupted or the heap is damaged.");
                 break;
            case  _HEAPBADBEGIN:
                  puts("The heap specified is not valid.");
                 break;
        }
        exit(rc);
    }
    free(ptr);
    return  0;

    /*****************************************************************************
         The output should be similar to :

         A memory node is corrupted or the heap is damaged.
    *****************************************************************************/
}
```

"Managing Memory" in the *Programming Guide*
"_uheapchk — Validate Memory Heap" on page 665
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heapset — Validate and Set Default Heap" on page 296
"_heap_walk — Return Information about Default Heap" on page 298
"<malloc.h>" on page 769
"<umalloc.h>" on page 779

_heapmin — Release Unused Memory from Default Heap

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
int _heapmin(void);
```

Description

Language Level: Extension

_heapmin returns all unused memory from the default runtime heap to the operating system.

Heap-specific, debug versions of this function (_uheapmin, and _debug_heapmin) are also available. _heapmin always operates on the default heap.

Note: If you create your own heap and make it the default heap, _heapmin calls the release function that you provide to return the memory.

Return Value

_heapmin returns 0 if successful; if not, it returns -1.

This example shows how to use the _heapmin function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   if  (_heapmin())
        printf("_heapmin  failed.\n");
   else
         printf("_heapmin was successful.\n");
   return  0;

   /*************************************************************************
        The output should be:

      _heapmin was successful.
   *************************************************************************/
}
```

Managing Memory in the *Programming Guide*

"_debug_heapmin — Release Unused Memory in the Default Heap" on page 128
"_debug_uheapmin — Release Unused Memory in User Heap" on page 136
"_uheapmin — Release Unused Memory in User Heap" on page 667
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_heapset — Validate and Set Default Heap

Format

#include <malloc.h>
int _heapset(unsigned int *fill*);

Description

Language Level: Extension

_heapset checks the default storage heap for minimal consistency by checking all allocated and freed objects on the heap (similar to _heapchk). It then sets each byte of the heap's free objects to the value of *fill*.

Using _heapset can help you locate problems where your program continues to use a freed pointer to an object. After you set the free heap to a specific value, when your program tries to interpret the set values in the freed object as data, unexpected results occur, indicating a problem.

A heap-specific version of this function, _uheapset, is also available.

Note: Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.

Return Value

_heapset returns one of the following values, defined in <malloc.h>:

_HEAPBADBEGIN The heap specified is not valid; it may have been closed or destroyed.
_HEAPBADNODE A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY The heap has not been initialized.
_HEAPOK The heap appears to be consistent.

This example allocates and frees memory, then uses _heapset to set the free heap to X. It checks the return code from _heapset to ensure the heap is still valid.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>

int main (void)
{
    char    *ptr;
    int     rc;

     if (NULL == (ptr = (char*)malloc(10))) {
          puts("Could not allocate memory block.");
          exit(EXIT_FAILURE);
    }
    memset(ptr,'A',5);
    free(ptr);

     if (_HEAPOK != (rc = _heapset('X'))) {
          switch(rc)  {
               case  _HEAPEMPTY:
                     puts("The heap has not been initialized.");
                     break;
               case  _HEAPBADNODE:
                     puts("A memory node is corrupted or the heap is damaged.");
                     break;
               case  _HEAPBADBEGIN:
                     puts("The heap specified is not valid.");
                     break;
          }
          exit(rc);
    }
    return  0;
}
```

"Managing Memory" in the *Programming Guide*
"_heapchk — Validate Default Memory Heap" on page 293
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heap_walk — Return Information about Default Heap" on page 298
"_uheapset — Validate and Set Memory Heap" on page 669
"<malloc.h>" on page 769
"<umalloc.h>" on page 779

_heap_walk — Return Information about Default Heap

Format

```
#include <malloc.h>
int _heap_walk(int (*callback_fn)(const void *object, size_t size,
                                                      int flag, int status,
                                                      const char* file, int line) );
```

Description

Language Level: Extension

_heap_walk traverses the default heap, and for each allocated or freed object, it calls the *callback_fn* function that you provide. For each object, it passes your function:

object A pointer to the object.

size The size of the object.

flag The value _USEDENTRY if the object is currently allocated, or _FREEENTRY if the object has been freed. (Both values are defined in <malloc.h>.)

status One of the following values, defined in <malloc.h>, depending on the status of the object:

- **_HEAPBADBEGIN** The heap specified is not valid; it may have been closed or destroyed.
- **_HEAPBADNODE** A memory node is corrupted, or the heap is damaged.
- **_HEAPEMPTY** The heap has not been initialized.
- **_HEAPOK** The heap appears to be consistent.

file The name of the file where the object was allocated.

line The line where the object was allocated.

_heap_walk provides information about all objects, regardless of which memory management functions were used to allocate them. However, the *file* and *line* information are only available if the object was allocated and freed using the debug versions of the memory management functions. Otherwise, *file* is NULL and *line* is 0.

_heap_walk calls *callback_fn* for each object until one of the following occurs:

All objects have been traversed.
callback_fn returns a nonzero value to _heap_walk.
It cannot continue because of a problem with the heap.

You may want to code your *callback_fn* to return a nonzero value if the status of the object is not _HEAPOK. Even if *callback_fn* returns 0 for an object that is corrupted, _heap_walk cannot continue because of the state of the heap and returns to its caller.

You can use *callback_fn* to process the information from _heap_walk in various ways. For example, you may want to print the information to a file or use it to generate your own error messages. You can use the information to look for memory leaks and objects incorrectly allocated or freed from the heap. It can be especially useful to call _heap_walk when _heapchk returns an error.

A heap-specific version of this function, _uheap_walk, is also available.

Notes:

1. Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.
2. _heap_walk locks the heap while it traverses it, to ensure that no other operations use the heap until _heap_walk finishes. As a result, in your *callback_fn*, you cannot call any critical functions in the runtime library, either explicitly or by calling another function that calls a critical function. See the *Programming Guide* for a list of critical functions.

Return Value

_heap_walk returns the last value of *status* to the caller.

This example allocates some memory, then uses _heap_walk to walk the heap and pass information about allocated objects to the callback function callback_function. callback_function then checks the information and keeps track of the number of allocated objects.

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

int _Optlink callback_function(const void *pentry, size_t sz, int useflag,
                               int status, const char *filename, size_t line)
{
    if (_HEAPOK != status) {
        puts("status is not _HEAPOK.");
        exit(status);
    }
```

```
    if (_USEDENTRY == useflag)
        printf("allocated   %p        %u\n", pentry, sz);
    else
        printf("freed       %p        %u\n",  pentry,  sz);
    return  0;
}

int main(void)
{
    char    *p1, *p2, *p3;

    if (NULL == (p1 = (char*)malloc(100)) ||
        NULL == (p2 = (char*)malloc(200)) ||
        NULL == (p3 = (char*)malloc(300))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    free(p2);
    puts("usage          address     size\n-----          -------     ----");

    _heap_walk(callback_function);

    free(p1);
    free(p3);
    return  0;

    /*************************************************************************
        The output should be similar to :

        usage          address     size
        -----          -------     ----
        allocated   73A10        300
        allocated   738B0        100
          :
          :
        freed       73920        224
    *************************************************************************/
}
```

"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"_heapchk — Validate Default Memory Heap" on page 293
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heapset — Validate and Set Default Heap" on page 296
"_uheap_walk — Return Information about Memory Heap" on page 671
"<malloc.h>" on page 769

hypot — Calculate Hypotenuse

Format

```
#include <math.h>
double hypot(double side1, double side2);
```

Description

Language Level: SAA, XPG4

hypot calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides *side1* and *side2*. A call to hypot is equivalent to:

```
sqrt(side1 * side1 + side2 * side2);
```

Return Value

hypot returns the length of the hypotenuse. If an overflow results, hypot sets errno to ERANGE and returns the value HUGE_VAL. If an underflow results, hypot sets errno to ERANGE and returns zero.

This example calculates the hypotenuse of a right-angled triangle with sides of 3.0 and 4.0.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y,z;

    x = 3.0;
    y = 4.0;
    z = hypot(x, y);
    printf("The hypotenuse of the triangle with sides %lf and %lf"
    " is %lf\n", x, y, z);
   return  0;

   /*****************************************************************************
        The output should be:

      The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
   *****************************************************************************/
}
```

"_fsqrt — Calculate Square Root" on page 254
"sqrt — Calculate Square Root" on page 527
"<math.h>" on page 770

iconv — Convert Characters to New Code Set

Format

```
#include <iconv.h>
size_t iconv(iconv_t cd, char **inbuf, size_t *inbytesleft,
                    char **outbuf, size_t *outbytesleft);
```

Description

Language Level: XPG4

iconv converts a sequence of characters in one encoded character set, in the array indirectly pointed to by *inbuf*, into a sequence of corresponding characters in the another encoded character set. It stores the corresponding sequence in the array indirectly pointed to by *outbuf*. *cd* is the conversion descriptor returned from iconv_open that describes which codesets are used in the conversion.

inbuf points to a variable that points to the first character in the input buffer, and *inbytesleft* indicates the number of bytes to the end of the buffer. *outbuf* points to a variable that points to the first available byte in the output buffer, and *outbytesleft* indicates the number of available bytes to the end of the buffer.

As iconv converts the characters, it updates the variable pointed to by *inbuf* to point to the next byte in the input buffer, and updates the variable pointed to by *outbuf* to the byte following the last successfully converted character. It also decrements *inbytesleft* and *outbytesleft* to reflect the number of bytes of input remaining and the number of bytes still available for output, respectively. If the entire input string is converted, *insize* will be 0; if an error stops the conversion, *insize* will have a nonzero value.

If it encounters a valid input character that does not have a defined conversion in *cd*, iconv translates the character to the ASCII value 0x1a.

Sharing a conversion descriptor in iconv across multiple threads may result in undefined behavior.

When the target *outbuf* is too small for the conversion, the remaining converted bytes stay within the run-time, internal buffer, and are processed by the next iconv call.

Return Value

iconv returns the number of nonidentical conversions performed. In a nonidentical conversion there is no equivalent character image in the target code pages. A substitution is required and the return code counts the number of instances when this happens.

If the entire string in the input buffer is converted, *inbytesleft* points to 0. If the input conversion stops, *inbytesleft* points to nonzero and **errno** is set to indicate

the condition that caused the conversion to stop. If an error occurs, iconv returns (size_t)-1, and sets **errno** to one of the following values:

Value	Meaning
EILSEQ	An input byte does not belong to the input codeset.
E2BIG	*outbuf* is not large enough to hold the converted value.
EINVAL	Incomplete character or shift sequence at the end of the input buffer.
EBADF	*cd* is not a valid conversion descriptor.

This example converts an array of characters coded in encoded character set IBM-850 (in in) to an array of characters coded in encoded character set IBM-037 (stored in out).

```
#include <iconv.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    const char fromcode[] = "IBM-850";
    const char tocode[] = "IBM-037";
   iconv_t       cd;
    char           in[] = "ABCDEabcde";
   size_t        in_size;
    char           *inptr = in;
   char           out[100];
    size_t        out_size = sizeof(out);
    char           *outptr = out;
   int            i;

    if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
        printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
       exit(EXIT_FAILURE);
   }
```

```
    /* Convert the first 3 characters in array "in". */
    in_size = 3;
    if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
         printf("Fail to convert first 3 characters to new code set.\n");
        exit(EXIT_FAILURE);
    }
    /* Convert the rest of the characters in array "in". */
    in_size = sizeof(in) - 3;
    if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
         printf("Fail to convert the rest of the characters to new code set.\n");
        exit(EXIT_FAILURE);
    }
    *outptr = '\0';
    printf("The hex representation of string %s are:\n    In codepage %s ==> ",
                in,  fromcode);
    for (i = 0; in[i] != '\0'; i++) {
         printf("0x%02x ", in[i]);
    }
    printf("\n    In codepage %s ==> ", tocode);
    for (i = 0; out[i] != '\0'; i++) {
         printf("0x%02x ", out[i]);
    }
    if (-1 == iconv_close(cd)) {
         printf("Fail to iconv_close.\n");
        exit(EXIT_FAILURE);
    }
   return  0;

   /*************************************************************************
         The output should be similar to :

         The hex representation of string ABCDEabcde are:
            In codepage IBM-850 ==> 0x41 0x42 0x43 0x44 0x45 0x61 0x62 0x63 0x64 0x65
            In codepage IBM-037 ==> 0xc1 0xc2 0xc3 0xc4 0xc5 0x81 0x82 0x83 0x84 0x85
   *************************************************************************/
}
```

"iconv_close — Remove Conversion Descriptor" on page 305
"iconv_open — Create Conversion Descriptor" on page 306
"setlocale — Set Locale" on page 499
"Introduction to Locale" in the *Programming Guide*
"<locale.h>" on page 766

iconv_close — Remove Conversion Descriptor

Format

```
#include <iconv.h>
int iconv_close(iconv_t cd);
```

Description

Language Level: XPG4

iconv_close deallocates the conversion descriptor *cd* and all other associated resources allocated by the iconv_open function.

Returned Value

If successful, iconv_close returns 0. Otherwise, iconv_close returns −1 is returned and sets **errno** to indicate the cause of the error. If *cd* is not a valid descriptor, an error occurs and iconv_close sets **errno** to EBADF.

This example shows how you would use iconv_close to remove a conversion descriptor.

```
#include <iconv.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    const char     fromcode[] = "IBM-850";
    const char     tocode[] = "IBM-863";
   iconv_t         cd;

    if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
        printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
       exit(EXIT_FAILURE);
   }
    if (-1 == iconv_close(cd)) {
        printf("Fail to iconv_close.\n");
       exit(EXIT_FAILURE);
   }
   return  0;
}
```

"iconv — Convert Characters to New Code Set" on page 302
"iconv_open — Create Conversion Descriptor" on page 306
"setlocale — Set Locale" on page 499
"Introduction to Locale" in the *Programming Guide*
"<locale.h>" on page 766

iconv_open — Create Conversion Descriptor

Format

#include <iconv.h>
iconv_t iconv_open(const char **tocode*, const char **fromcode*);

Description

Language Level: XPG4

iconv_open performs all the initialization needed to convert characters from the encoded character set specified in the array pointed to by *fromcode* to the encoded character set specified in the array pointed to by *tocode*. It creates a conversion descriptor that relates the two encoded character sets. You can then use the conversion descriptor with the iconv function to convert characters between the codesets.

The conversion descriptor remains valid until you close it with iconv_close.

For information about the settings of *fromcode*, *tocode*, and their permitted combinations, see the section on internationalization in the *Programming Guide*.

Returned Value

If successful, iconv_open returns a conversion descriptor of type iconv_t. Otherwise, it returns (iconv_t)–1, and sets **errno** to indicate the error. If you cannot convert between the encoded character sets specified, an error occurs and iconv_open sets **errno** to EINVAL.

This example shows how to use iconv_open, iconv, and iconv_close to convert characters from one codeset to another.

```
#include <iconv.h>
#include <stdlib.h>
#include <stdio.h>
```

```
int main(void)
{
    const char     fromcode[] = "IBM-932";
    const char     tocode[] = "IBM-930";
   iconv_t         cd;
    char             in[] = "\x81\x40\x81\x80";
    size_t          in_size = sizeof(in);
    char             *inptr = in;
   char             out[100];
    size_t          out_size = sizeof(out);
    char             *outptr = out;
   int              i;

    if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
        printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
       exit(EXIT_FAILURE);
   }
    if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
        printf("Fail to convert characters to new code set.\n");
       exit(EXIT_FAILURE);
   }
    *outptr = '\0';
    printf("The hex representation of string %s are:\n    In codepage %s ==> ",
             in,  fromcode);
    for (i = 0; in[i] != '\0'; i++) {
        printf("0x%02x ", in[i]);
   }
    printf("\n    In codepage %s ==> ", tocode);
    for (i = 0; out[i] != '\0'; i++) {
        printf("0x%02x ", out[i]);
   }
    if (-1 == iconv_close(cd)) {
        printf("Fail to iconv_close.\n");
       exit(EXIT_FAILURE);
   }
   return  0;

   /**************************************************************************
        The output should be similar to :

        The hex representation of string  @ ä are:
           In codepage IBM-932 ==> 0x81 0x40 0x81 0x80
           In codepage IBM-930 ==> 0x0e 0x40 0x40 0x44 0x7b 0x0f
   **************************************************************************/
}
```


"iconv — Convert Characters to New Code Set" on page 302
"iconv_close — Remove Conversion Descriptor" on page 305
"setlocale — Set Locale" on page 499
"Introduction to Locale" in the *Programming Guide*
"<locale.h>" on page 766

_inp — Read Byte from Input Port

Format

```
#include <conio.h>    /* also in <builtin.h>    */
int _inp(const unsigned int port);
```

Description

Language Level: Extension

_inp reads a byte from the specified input *port*. The *port* number must be an unsigned int value in the range 0 to 65 535 inclusive.

Note: _inp is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _inp.
You cannot parenthesize a call to _inp. (Parentheses specify a call to the function's backing code, and _inp has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_inp returns the byte value as an integer that was read from the specified *port*. There is no error return value, and _inp does not set errno.

This example uses _inp to read a byte from a specified input port and return the data read.

```
#include <builtin.h>

#define LOWER 0
#define UPPER 65535

int Add1(int j);

int      g;
enum     fruit {apples=10, bananas, cantaloupes};
int      arr[] = {cantaloupes, bananas, apples};
struct
{
    int    i;
    char  ch;
} st;
```

```
int main(void)
{
    int  i;
     volatile const int c = 0;

     i = _inp(0);
     g = _inp(LOWER + 1);       /* put the data read in a global variable    */
     i = _inp(apples);           /* passing enumerated type as the            */
                                     /* port number                                  */
                                    /*  ===================================    */
                                     /* Types of port number passed :              */
                                    /*  ----------------------------------    */
     i = _inp(c);                  /* - constant                                   */
     i = _inp(arr[c]);            /* - element of array                          */
     st.i = Add1(c);               /*                                              */
     i = _inp(st.i);              /* - element of structure                      */
     i = _inp(Add1(LOWER));        /* - return value from a function call    */
     i = _inp(UPPER);              /* - #define constant                         */
     i = _inp(256);                /* - the exact port number                     */
                                    /*  ----------------------------------    */
    return  0;
}

int Add1(int j)
{
     j += 1;
    return  j;
}
```

"_inpw — Read Unsigned Short from Input Port" on page 312
"_inpd — Read Doubleword from Input Port" on page 310
"isatty — Test Handle for Character Device" on page 321
"_outp — Write Byte to Output Port" on page 421
"_outpw — Write Word to Output Port" on page 425
"_outpd — Write Double Word to Output Port" on page 423
"<builtin.h>" on page 761
"<conio.h>" on page 762

_inpd — Read Doubleword from Input Port

Format

```
#include <conio.h>    /* also in <builtin.h> */
unsigned long _inpd(const unsigned int port);
```

Description

Language Level: Extension

_inpd reads a 4-byte (doubleword) unsigned value from the specified input *port*. The *port* number must be an unsigned short value within the range 0 to 65 535 inclusive.

Note: _inpd is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _inpd.
You cannot parenthesize a call to _inpd. (Parentheses specify a call to the function's backing code, and _inpd has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_inpd returns the value read from the specified *port*. There is no error return value, and _inpd does not set errno.

This example uses _inpd to read a doubleword value from the specified input port and return the data read.

```
#include <builtin.h>

#define LOWER 0
#define UPPER 65535

int Add1(int j);
```

```
static long g;
enum            fruit {apples=10, bananas, cantaloupes};
int             arr[] = {cantaloupes, bananas, apples};
union
{
    int    i;
    char  ch;
} un;

int main(void)
{
    unsigned  long          l;
     volatile const int c = 0;

     un.i = 65534;
     g = _inpd(255);              /* put the data read in a global variable    */
                                                  /* ============================ */
                                                  /* Types of port number passed:*/
                                                  /* --------------------------- */
     l = _inpd(c);                              /* - constant                          */
     l = _inpd(un.i);                          /* - element of union                 */
     l = _inpd(Add1(cantaloupes));            /* - return value from a            */
                                                  /*     function  call                  */
     l = _inpd(_inp(arr[Add1(LOWER)]));    /* - return value from a            */
                                                  /*     builtin function call         */
                                                  /* --------------------------- */
    return  0;
}

int Add1(int j)
{
     j += 1;
    return  j;
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpw — Read Unsigned Short from Input Port" on page 312
"isatty — Test Handle for Character Device" on page 321
"_outp — Write Byte to Output Port" on page 421
"_outpw — Write Word to Output Port" on page 425
"_outpd — Write Double Word to Output Port" on page 423
"<builtin.h>" on page 761
"<conio.h>" on page 762

_inpw — Read Unsigned Short from Input Port

Format

```
#include <conio.h>    /* also in <builtin.h> */
unsigned short _inpw(const unsigned int port);
```

Description

Language Level: Extension

_inpw reads an unsigned short value from the specified input *port*. The *port* number must be an unsigned short value within the range 0 to 65 535 inclusive.

Note: _inpw is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _inpw.
You cannot parenthesize a call to _inpw. (Parentheses specify a call to the function's backing code, and _inpw has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_inpw returns the value read from the specified *port*. There is no error return value, and _inpw does not set errno.

This example uses _inpw to read an unsigned short value from the specified input port and return the data read.

```
#include <builtin.h>

#define LOWER 0
#define UPPER 65535

int Add1(int j);

volatile short g;
```

```
int main(void)
{
    volatile unsigned short s;
    volatile const int c = 0;
    int i = 65534;
    enum fruit {apples, bananas, cantaloupes};
    int arr[] = {cantaloupes, bananas, apples};

    g = _inpw(LOWER);          /* put the data read in a global variable   */
    s = _inpw(bananas);                    /* passing enumerated type        */
                                              /* as the port number              */
                                              /* ========================== */
                                              /* Types of port number passed:*/
                                              /* -------------------------- */
    s = _inpw(c);                          /* - constant                        */
    s = _inpw(i+1);                        /* - integer                         */
    s = _inpw(arr[bananas]);               /* - element of array                */
    s = _inpw(_outp(UPPER,cantaloupes));/* - return value from a             */
                                              /*     builtin function call       */
                                              /* -------------------------- */
   return  0;
}

int Add1(int j)
{
    j += 1;
   return  j;
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpd — Read Doubleword from Input Port" on page 310
"isatty — Test Handle for Character Device" on page 321
"_outp — Write Byte to Output Port" on page 421
"_outpw — Write Word to Output Port" on page 425
"_outpd — Write Double Word to Output Port" on page 423
"<builtin.h>" on page 761
"<conio.h>" on page 762

_interrupt — Call Interrupt Procedure

Format

```
#include <builtin.h>
void _interrupt(const unsigned int intnum);
```

Description

Language Level: Extension

_interrupt calls the interrupt procedure specified by *intnum* using the INT machine instruction. The integer *intnum* must have a value within the range 0 to 255 inclusive.

Note: _interrupt is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _interrupt.
You cannot parenthesize a call to _interrupt. (Parentheses specify a call to the function's backing code, and _interrupt has none.)

Return Value

There is no return value, and _interrupt does not set errno.

This example calls interrupt 3, which is a breakpoint.

```
#include <builtin.h>

int main(void)
{
    /* A breakpoint will occur when running this program */
    /* within a debugger.                                */
   _interrupt(3);

   return  0;
}
```

"_disable — Disable Interrupts" on page 142
"_enable — Enable Interrupts" on page 168
"<builtin.h>" on page 761

isalnum **to** isxdigit **— Test Integer Value**

Format

```
#include <ctype.h>
                                        /* test for:                    */
int isalnum(int c);   /* alphanumeric character */
int isalpha(int c);   /* alphabetic character     */
int iscntrl(int c);   /* control character        */
int isdigit(int c);   /* decimal digit            */
int isgraph(int c);   /* printable character, excluding space */
int islower(int c);   /* lowercase character      */
int isprint(int c);   /* printable character, including space */
int ispunct(int c);   /* nonalphanumeric printable character, excluding space */
int isspace(int c);   /* whitespace character     */
int isupper(int c);   /* uppercase character      */
int isxdigit(int c); /* hexadecimal digit        */
```

Description **Language Level:** ANSI, SAA, POSIX, XPG4

These functions test a given integer value *c* to determine if it has a certain property as defined by the LC_CTYPE category of your current locale. The value of *c* must be representable as an **unsigned char**, or EOF.

The functions test for the following:

isalnum — Alphanumeric character (upper- or lowercase letter, or decimal digit), as defined in the locale source file in the alnum class of the LC_CTYPE category of the current locale.

isalpha — Alphabetic character, as defined in the locale source file in the alpha class of the LC_CTYPE category of the current locale.

iscntrl — Control character, as defined in the locale source file in the cntrl class of the LC_CTYPE category of the current locale.

isdigit — Decimal digit (0 through 9), as defined in the locale source file in the digit class of the LC_CTYPE category of the current locale.

isgraph — Printable character, excluding the space character, as defined in the locale source file in the graph class of the LC_CTYPE category of the current locale.

islower — Lowercase letter, as defined in the locale source file in the lower class of the LC_CTYPE category of the current locale.

isprint — Printable character, including the space character, as defined in the locale source file in the print class of the LC_CTYPE category of the current locale.

ispunct Nonalphanumeric printable character, excluding the space character, as defined in the locale source file in the punct class of the LC_CTYPE category of the current locale.

isspace White-space character, as defined in the locale source file in the space class of the LC_CTYPE category of the current locale.

isupper Uppercase letter, as defined in the locale source file in the upper class of the LC_CTYPE category of the current locale.

isxdigit Hexadecimal digit (0 through 9, a through f, or A through F), as defined in the locale source file in the xdigit class of the LC_CTYPE category of the current locale.

You can redefine any character class in the LC_CTYPE category of the current locale, with some restrictions. See the section about the LC_CTYPE class in the *Programming Guide* for details about these restrictions.

Return Value These functions return a nonzero value if the integer satisfies the test condition, or 0 if it does not.

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255, indicating whether they have the properties tested for.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <locale.h>

#define UPPER_LIMIT     0xFF
#if (1 == __TOS_OS2__)
    #define LOCNAME "en_us.ibm-437"        /* OS/2 name             */
#else
    #define LOCNAME "en_us.ibm-1252"       /* Windows name          */
#endif

int main(void)
{
   int  ch;
```

```
 if (NULL == setlocale(LC_ALL, LOCNAME)) {
      printf("Locale \"%s\" could not be loaded\n", LOCNAME);
     exit(1);
}
 for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
      printf("%#04x ", ch);
      printf("%c", isprint(ch)    ? ch          : ' ');
      printf("%s", isalnum(ch)    ? " AN"    : "      ");
      printf("%s", isalpha(ch)    ? " A "    : "      ");
      printf("%s", iscntrl(ch)    ? " C "    : "      ");
      printf("%s", isdigit(ch)    ? " D "    : "      ");
      printf("%s", isgraph(ch)    ? " G "    : "      ");
      printf("%s", islower(ch)    ? " L "    : "      ");
      printf("%s", ispunct(ch)    ? " PU"    : "      ");
      printf("%s", isspace(ch)    ? " S "    : "      ");
      printf("%s", isprint(ch)    ? " PR"    : "      ");
      printf("%s", isupper(ch)    ? " U "    : "      ");
      printf("%s", isxdigit(ch) ? " H "    : "      ");
     putchar('\n');
}
return  0;

/*****************************************************************************
      The output should be similar to :
     :
     0x20                                  S    PR
     0x21  !                  G        PU       PR
     0x22  "                  G        PU       PR
     0x23  #                  G        PU       PR
     0x24  $                  G        PU       PR
     0x25  %                  G        PU       PR
     0x26  &                  G        PU       PR
     0x27  '                  G        PU       PR
     0x28  (                  G        PU       PR
     0x29  )                  G        PU       PR
     0x2a  *                  G        PU       PR
     0x2b  +                  G        PU       PR
     0x2c  ,                  G        PU       PR
     0x2d  -                  G        PU       PR
     0x2e  .                  G        PU       PR
     0x2f  /                  G        PU       PR
```

```
    0x30 0 AN       D   G           PR      H   H
    0x31 1 AN       D   G           PR      H   H
    0x32 2 AN       D   G           PR      H   H
    0x33 3 AN       D   G           PR      H   H
    0x34 4 AN       D   G           PR      H   H
    0x35 5 AN       D   G           PR      H   H
    0x36 6 AN       D   G           PR      H   H
    0x37 7 AN       D   G           PR      H   H
    0x38 8 AN       D   G           PR      H   H
    0x39 9 AN       D   G           PR      H   H
    :
  ****************************************************************************/
}
```

"isascii — Test Integer Values" on page 319
"_iscsym - _iscsymf — Test Integer" on page 325
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"setlocale — Set Locale" on page 499
"tolower - toupper — Convert Character Case" on page 631
"_toascii - _tolower - _toupper — Convert Character" on page 628
"<ctype.h>" on page 762

isascii — Test Integer Values

Format

```
#include <ctype.h>
int isascii(int c);
```

Description

Language Level: XPG4, Extension

isascii tests if an integer is within the ASCII range. This macro assumes that the system uses the ASCII character set.

Note: In earlier releases of VisualAge C++, isascii began with an underscore (_isascii). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _isascii to isascii for you.

Return Value

isascii returns a nonzero value if the integer is within the ASCII set, and 0 if it is not.

This example tests the integers from 0x7c to 0x82, and prints the corresponding ASCII character if the integer is within the ASCII range.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int  ch;

    for (ch = 0x7c; ch <= 0x82; ch++) {
       printf("%#04x       ",  ch);
       if  (isascii(ch))
            printf("The ASCII character is %c\n", ch);
       else
            printf("Not an ASCII character\n");
   }
   return  0;

   /*************************************************************************
        The output should be:

        0x7c       The ASCII character is |
        0x7d       The ASCII character is }
        0x7e       The ASCII character is
        0x7f       The ASCII character is '
        0x80       Not an ASCII character
        0x81       Not an ASCII character
        0x82       Not an ASCII character
   *************************************************************************/
}
```

isascii

"isalnum to isxdigit — Test Integer Value" on page 315
"_iscsym - _iscsymf — Test Integer" on page 325
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"_toascii - _tolower - _toupper — Convert Character" on page 628
"tolower - toupper — Convert Character Case" on page 631
"<ctype.h>" on page 762

isatty — Test Handle for Character Device

Format

#include <io.h>
int isatty(int *handle*);

Description

Language Level: XPG4, Extension

isatty determines whether the given handle is associated with a character device (a keyboard, display, or printer or serial port).

Note: In earlier releases of VisualAge C++, isatty began with an underscore (_isatty). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _isatty to isatty for you.

Return Value

isatty returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

This example opens the console and determines if it is a character device:

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int  fh;

    if (-1 == (fh = fileno(stdin))) {
        perror("Error getting file handle from stdin.\n");
       return  EXIT_FAILURE;
   }

    if (0 != isatty(fh))
        printf("stdin is a character device.\n");
   else
        printf("stdin is not a character device.\n");

   return  0;

   /*****************************************************************************
        The output should be:

        stdin is a character device.
   *****************************************************************************/
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpd — Read Doubleword from Input Port" on page 310
"_inpw — Read Unsigned Short from Input Port" on page 312
"_outp — Write Byte to Output Port" on page 421
"_outpd — Write Double Word to Output Port" on page 423
"_outpw — Write Word to Output Port" on page 425

isatty

"<io.h>" on page 764

isblank — Test for Blank Character Classification

Format

```
#include <ctype.h>
int isblank(int c);
```

Description

Language Level: Extension

isblank tests whether the current LC_CTYPE locale category assigns *c* the blank character attribute.

The value for *c* must be representable as an unsigned character, or EOF.

In the "POSIX" and "C" locales, the tab and space characters have the blank attribute.

Return Value

isblank returns a nonzero value if the integer *c* has the blank attribute; 0 if it does not.

This example tests if c is a blank type.

```
#include <ctype.h>
#include <locale.h>
#include <stdio.h>

void check(char c) {
     if ((' ' != c) && (isprint(c)))
          printf("    %c is ", c);
    else
          printf("x%02x is ", c);
    if  (!isblank(c))
         printf("not  ");
     puts("a blank type character");
    return;
}

int main(void)
{
     printf("In LC_CTYPE category of locale name \"%s\":\n",
               setlocale(LC_CTYPE,  NULL));
    check('a');
    check('  ');
    check(0x00);
    check('\n');
    check('\t');
    return  0;

    /*************************************************************************
          The output should be similar to :

          In LC_CTYPE category of locale name "C":
             a is not a blank type character
          x20 is a blank type character
          x00 is not a blank type character
```

```
    x0a is not a blank type character
    x09 is a blank type character
*****************************************************************************/
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"_iscsym - _iscsymf — Test Integer" on page 325
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"iswblank — Test for Wide Blank Character Classification" on page 331
"setlocale — Set Locale" on page 499
"<ctype.h>" on page 762

_iscsym - _iscsymf — Test Integer

Format

```
#include <ctype.h>
int _iscsym(int c);
int _iscsymf(int c);
```

Language Level: Extension

These macros test if an integer is within a particular ASCII set. The macros assume that the system uses the ASCII character set.

iscsym tests if a character is alphabetic, a digit, or an underscore (). _iscsymf tests if a character is alphabetic or an underscore.

Return Value

_iscsym and _iscsymf return a nonzero value if the integer is within the ASCII set for which it tests, and 0 if it is not.

This example uses _iscsym and _iscsymf to test the characters a, _, and 1. If the character falls within the ASCII set tested for, the macro returns TRUE. Otherwise, it returns FALSE.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch[3] =    { 'a','_','1' };
   int  i;

    for (i = 0; i < 3; i++) {
        printf("_iscsym('%c') returns %s\n", ch[i], _iscsym(ch[i])?"TRUE":"FALSE");
        printf("_iscsymf('%c') returns %s\n\n", ch[i], _iscsymf(ch[i])?"TRUE":
            "FALSE");
   }
   return  0;

   /*****************************************************************************
        The output should be:

        _iscsym('a') returns TRUE
        _iscsymf('a') returns TRUE

        _iscsym('_') returns TRUE
        _iscsymf('_') returns TRUE

        _iscsym('1') returns TRUE
        _iscsymf('1') returns FALSE
   *****************************************************************************/
}
```

_iscsym – _iscsymf

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"isblank — Test for Blank Character Classification" on page 323
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"iswblank — Test for Wide Blank Character Classification" on page 331
"tolower - toupper — Convert Character Case" on page 631
"_toascii - _tolower - _toupper — Convert Character" on page 628
"<ctype.h>" on page 762

iswalnum to iswxdigit — Test Wide Integer Value

Format

```
#include <wctype.h>
                                        /* test for:         */
int iswalnum(wint_t wc);   /* wide alphanumeric character */
int iswalpha(wint_t wc);   /* wide alphabetic character   */
int iswcntrl(wint_t wc);   /* wide control character      */
int iswdigit(wint_t wc);   /* wide decimal digit          */
int iswgraph(wint_t wc);   /* wide printable character, excluding space */
int iswlower(wint_t wc);   /* wide lowercase character    */
int iswprint(wint_t wc);   /* wide printable character, including space */
int iswpunct(wint_t wc);   /* wide punctuation character, excluding space */
int iswspace(wint_t wc);   /* wide whitespace character   */
int iswupper(wint_t wc);   /* wide uppercase character    */
int iswxdigit(wint_t wc); /* wide hexadecimal digit       */
```

Description

Language Level: ANSI 93, POSIX, XPG4

These functions test a given wide integer value *wc* to determine whether it has a certain property as defined by the LC_CTYPE category of your current locale. The value of *wc* must be representable as a wint_t.

The functions test for the following:

iswalnum — Wide alphanumeric character (upper- or lowercase letter, or decimal digit), as defined in the locale source file in the alnum class of the LC_CTYPE category of the current locale.

iswalpha — Wide alphabetic character, as defined in the locale source file in the alpha class of the LC_CTYPE category of the current locale.

iswcntrl — Wide control character, as defined in the locale source file in the cntrl class of the LC_CTYPE category of the current locale.

iswdigit — Wide decimal digit (0 through 9), as defined in the locale source file in the digit class of the LC_CTYPE category of the current locale.

iswgraph — Wide printable character, excluding the space character, as defined in the locale source file in the graph class of the LC_CTYPE category of the current locale.

iswlower — Wide lowercase letter, as defined in the locale source file in the lower class of the LC_CTYPE category of the current locale.

iswprint — Wide printable character, including the space character, as defined in the locale source file in the print class of the LC_CTYPE category of the current locale.

iswpunct Wide non-alphanumeric printable character, excluding the space character, as defined in the locale source file in the punct class of the LC_CTYPE category of the current locale.

iswspace Wide white-space character, as defined in the locale source file in the space class of the LC_CTYPE category of the current locale.

iswupper Wide uppercase letter, as defined in the locale source file in the upper class of the LC_CTYPE category of the current locale.

iswxdigit Wide hexadecimal digit (0 through 9, a through f, or A through F), as defined in the locale source file in the xdigit class of the LC_CTYPE category of the current locale.

You can redefine any character class in the LC_CTYPE category of the current locale. For more information, see "Introduction to Locales" in the *Programming Guide*.

Return Value These functions return a nonzero value if the wide integer satisfies the test value; 0 if it does not.

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255, indicating whether they have the properties tested for.

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>
#include <wctype.h>

#define UPPER_LIMIT     0xFF
#if (1 == __TOS_OS2__)
    #define LOCNAME "en_us.ibm-437"         /* OS/2 name              */
#else
    #define LOCNAME "en_us.ibm-1252"        /* Windows name           */
#endif

int main(void)
{
   wint_t  wc;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    for (wc = 0; wc <= UPPER_LIMIT; wc++) {
        printf("%#4x ", wc);
        printf("%c", iswprint(wc)    ? wc        : ' ');
        printf("%s", iswalnum(wc)    ? " AN" : "      ");
        printf("%s", iswalpha(wc)    ? " A " : "      ");
        printf("%s", iswcntrl(wc)    ? " C " : "      ");
        printf("%s", iswdigit(wc)    ? " D " : "      ");
        printf("%s", iswgraph(wc)    ? " G " : "      ");
        printf("%s", iswlower(wc)    ? " L " : "      ");
        printf("%s", iswpunct(wc)    ? " PU" : "      ");
        printf("%s", iswspace(wc)    ? " S " : "      ");
        printf("%s", iswprint(wc)    ? " PR" : "      ");
        printf("%s", iswupper(wc)    ? " U " : "      ");
        printf("%s", iswxdigit(wc) ? " H " : "      ");
       putchar('\n');
   }
   return  0;

   /*************************************************************************
        The output should be similar to :
       :
       0x20                                  S    PR
       0x21  !                    G        PU       PR
       0x22  "                    G        PU       PR
       0x23  #                    G        PU       PR
       0x24  $                    G        PU       PR
       0x25  %                     G       PU       PR
       0x26  &                     G       PU       PR
       0x27  '                    G        PU       PR
       0x28  (                    G        PU       PR
```

```
    0x29  )                G       PU      PR
    0x2a  *                G       PU      PR
    0x2b  +                G       PU      PR
    0x2c  ,                G       PU      PR
    0x2d  -                G       PU      PR
    0x2e  .                G       PU      PR
    0x2f  /                G       PU      PR
     0x30 0 AN          D   G              PR      H
     0x31 1 AN          D   G              PR      H
     0x32 2 AN          D   G              PR      H
     0x33 3 AN          D   G              PR      H
     0x34 4 AN          D   G              PR      H
     0x35 5 AN          D   G              PR      H
     0x36 6 AN          D   G              PR      H
     0x37 7 AN          D   G              PR      H
     0x38 8 AN          D   G              PR      H
     0x39 9 AN          D   G              PR      H
    :
  ****************************************************************************/
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"isblank — Test for Blank Character Classification" on page 323
"_iscsym - _iscsymf — Test Integer" on page 325
"iswblank — Test for Wide Blank Character Classification" on page 331
"iswctype — Test for Character Property" on page 333
"<wctype.h>" on page 782

iswblank — Test for Wide Blank Character Classification

Format

```
#include <wctype.h>
int iswblank(wint_t wc);
```

Description

Language Level: Extension

iswblank tests whether the current LC_CTYPE locale category assigns the the blank character attribute to the wide character *wc*.

The value for *wc* must be representable as a wchar_t, or WEOF.

The behavior of iswblank is affected by the LC_CTYPE category of the current locale.

In the "POSIX" and "C" locales, the tab and space characters have the blank attribute.

Return Value

iswblank returns a nonzero value if *wc* has the blank attribute; 0 if it does not.

This example tests whether wc is a blank type.

```
#include <stdio.h>
#include <wctype.h>
#include <wchar.h>
#include <locale.h>

void check(wchar_t wc) {
     if ((' ' != wc) && (iswprint(wc)))
          printf("    %lc is ", wc);
    else
          printf("x%02x is ", wc);
    if  (!iswblank(wc))
         printf("not  ");
     puts("a blank type character");
    return;
}

int main(void)
{
     printf("In LC_CTYPE category of locale name \"%s\":\n",
               setlocale(LC_CTYPE,  NULL));
    check(L'a');
    check(L'  ');
    check(0x00);
    check(L'\n');
    check(L'\t');
    return  0;

    /*************************************************************************
         The output should be similar to :
```

```
    In LC_CTYPE category of locale name "C":
       a is not a blank type character
    x20 is a blank type character
    x00 is not a blank type character
    x0a is not a blank type character
    x09 is a blank type character
*******************************************************************************/
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"isblank — Test for Blank Character Classification" on page 323
"_iscsym - _iscsymf — Test Integer" on page 325
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"iswctype — Test for Character Property" on page 333
"<wctype.h>" on page 782

iswctype — Test for Character Property

Format

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);
```

Description

Language Level: ANSI 93, XPG4

iswctype determines whether the wide character *wc* has the property *wc_prop*. It is similar in function to the iswalnum through isxdigit functions, but with iswctype you can specify the property to check for, or check for a property other than the standard ones.

You must obtain the *wc_prop* value from a call to wctype. If you do not, or if the LC_CTYPE category of the locale was modified after you called wctype, the behavior of iswctype is undefined.

The value of *wc* must be representable as an unsigned wchar_t, or WEOF.

The following strings correspond to the standard (basic) character classes or properties:

"alnum"	"cntrl"	"lower"	"space"
"alpha"	"digit"	"print"	"upper"
"blank"	"graph"	"punct"	"xdigit"

The following shows calls to wctype and indicates the equivalent isw* function:

```
iswctype(wc, wctype("alnum"));     /* iswalnum(wc);   */
iswctype(wc, wctype("alpha"));     /* iswalpha(wc);   */
iswctype(wc, wctype("blank"));     /* iswblank(wc);   */
iswctype(wc, wctype("cntrl"));     /* iswcntrl(wc);   */
iswctype(wc, wctype("digit"));     /* iswdigit(wc);   */
iswctype(wc, wctype("graph"));     /* iswgraph(wc);   */
iswctype(wc, wctype("lower"));     /* iswlower(wc);   */
iswctype(wc, wctype("print"));     /* iswprint(wc);   */
iswctype(wc, wctype("punct"));     /* iswpunct(wc);   */
iswctype(wc, wctype("space"));     /* iswspace(wc);   */
iswctype(wc, wctype("upper"));     /* iswupper(wc);   */
iswctype(wc, wctype("xdigit"));    /* iswxdigit(wc); */
```

Return Value

iswctype returns a nonzero value if the wide character has the property tested for. If the value for *wc* or *wc_prop* is not valid, the behavior is undefined.

iswctype

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255, indicating whether they have the properties tested for.

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>
#include <wctype.h>

#define UPPER_LIMIT      0xFF
#if (1 == __TOS_OS2__)
    #define LOCNAME "en_us.ibm-437"         /* OS/2 name              */
#else
    #define LOCNAME "en_us.ibm-1252"        /* Windows name           */
#endif

int main(void)
{
   wint_t  wc;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
         printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
   }
    for (wc = 0; wc <= UPPER_LIMIT; wc++) {
         printf("%#4x ", wc);
         printf("%c", iswctype(wc, wctype("print"))    ? wc       : ' ');
         printf("%s", iswctype(wc, wctype("alnum"))    ? " AN" : "      ");
         printf("%s", iswctype(wc, wctype("alpha"))    ? " A " : "      ");
         printf("%s", iswctype(wc, wctype("blank"))    ? " B " : "      ");
         printf("%s", iswctype(wc, wctype("cntrl"))    ? " C " : "      ");
         printf("%s", iswctype(wc, wctype("digit"))    ? " D " : "      ");
         printf("%s", iswctype(wc, wctype("graph"))    ? " G " : "      ");
         printf("%s", iswctype(wc, wctype("lower"))    ? " L " : "      ");
         printf("%s", iswctype(wc, wctype("punct"))    ? " PU" : "      ");
         printf("%s", iswctype(wc, wctype("space"))    ? " S " : "      ");
         printf("%s", iswctype(wc, wctype("print"))    ? " PR" : "      ");
         printf("%s", iswctype(wc, wctype("upper"))    ? " U " : "      ");
         printf("%s", iswctype(wc, wctype("xdigit")) ? " H " : "      ");
        putchar('\n');
   }
   return  0;
```

```
/*****************************************************************************
    The output should be similar to :
        .
        .
    0x1e                C
    0x1f                C
    0x20            B                       S   PR
    0x21  !                     G       PU      PR
    0x22  "                     G       PU      PR
    0x23  #                     G       PU      PR
    0x24  $                     G       PU      PR
    0x25  %                     G       PU      PR
        .
        .
    0x30 0 AN               D   G               PR      H
    0x31 1 AN               D   G               PR      H
    0x32 2 AN               D   G               PR      H
    0x33 3 AN               D   G               PR      H
    0x34 4 AN               D   G               PR      H
    0x35 5 AN               D   G               PR      H
        .
        .
    0x43 C AN A                 G               PR U    H
    0x44 D AN A                 G               PR U    H
    0x45 E AN A                 G               PR U    H
    0x46 F AN A                 G               PR U    H
    0x47 G AN A                 G               PR U
        .
        .
*****************************************************************************/
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"isblank — Test for Blank Character Classification" on page 323
"_iscsym - _iscsymf — Test Integer" on page 325
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"iswblank — Test for Wide Blank Character Classification" on page 331
"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"wctype — Get Handle for Character Property Classification" on page 755
"<wctype.h>" on page 782

_itoa — Convert Integer to String

Format

#include <stdlib.h>
char *_itoa(int *value*, char * *string*, int *radix*);

Description

Language Level: Extension

_itoa converts the digits of the given *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

Note: The space reserved for *string* must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

Return Value

_itoa returns a pointer to *string*. There is no error return value.

This example converts the integer value -255 to a decimal, a binary, and a hex number, storing its character representation in the array buffer.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  buffer[35];
   char  *p;

    p = _itoa(-255, buffer, 10);
    printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
    p = _itoa(-255, buffer, 2);
    printf("The result of _itoa(-255) with radix of 2\n       is %s\n", p);
    p = _itoa(-255, buffer, 16);
    printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
   return  0;

   /*****************************************************************************
       The output should be:

       The result of _itoa(-255) with radix of 10 is -255
       The result of _itoa(-255) with radix of 2
            is  11111111111111111111111100000001
       The result of _itoa(-255) with radix of 16 is ffffff01
   *****************************************************************************/
}
```

"_ecvt — Convert Floating-Point to Character" on page 166
"_fcvt — Convert Floating-Point to String" on page 192
"_gcvt — Convert Floating-Point to String" on page 268
"_ltoa — Convert Long Integer to String" on page 367
"_ultoa — Convert Unsigned Long Integer to String" on page 675
"<stdlib.h>" on page 775

_kbhit — Test for Keystroke

Format

```
include <conio.h>
int _kbhit(void);
```

Description

Language Level: Extension

_kbhit tests if a key has been pressed on the keyboard. If the result is nonzero, a keystroke is waiting in the buffer. You can read in the keystroke using the _getch or _getche function. By calling _kbhit prior to calling _getch or _getche, you can prevent your program from waiting for a keyboard input.

Return Value

_kbhit returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

This example uses _kbhit to test for the pressing of a key on the keyboard and to print a statement with the test result.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int  ch;

    printf("Type in some letters.\n");
    printf("If you type in an 'x', the program ends.\n");

    for ( ; ; ) {
        while (0 == _kbhit()) {
        /* Processing without waiting for a key to be pressed */
        }

        ch = _getch();
        printf("You have pressed the '%c' key.\n", ch);

        if ('x' == ch)
            break;
   }
   return  0;

   /*************************************************************************
        The output should be similar to:

        Type in some letters.
        If you type in an 'x', the program ends.
        You have pressed the 'f' key.
        You have pressed the 'e' key.
        You have pressed the 'l' key.
        You have pressed the 'i' key.
        You have pressed the 'x' key.
   *************************************************************************/
}
```

"_getch - _getche — Read Character from Keyboard" on page 272

_kbhit

"<conio.h>" on page 762

labs — Calculate Absolute Value of Long Integer

Format

```
#include <stdlib.h>
long int labs(long int n);
```

Description

Language Level: ANSI, SAA, XPG4

labs produces the absolute value of its long integer argument *n*. The result may be undefined when the argument is equal to LONG_MIN, the smallest available long integer (-2 147 483 647). The value LONG_MIN is defined in the <limits.h> include file.

Return Value

labs returns the absolute value of *n*. There is no error return value.

This example computes *y* as the absolute value of the long integer -41567.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long  x,y;

    x = -41567L;
    y = labs(x);
    printf("The absolute value of %ld is %ld\n", x, y);
   return  0;

   /*************************************************************************
        The output should be:

        The absolute value of -41567 is 41567
   *************************************************************************/
}
```

"abs — Calculate Integer Absolute Value" on page 40
"_cabs — Calculate Absolute Value of Complex Number" on page 73
"fabs — Calculate Floating-Point Absolute Value" on page 182
"<limits.h>" on page 766
"llabs — Calculate Absolute Value of Long Long Integer" on page 344

ldexp — Multiply by a Power of Two

Format

```
#include <math.h>
double ldexp(double x, int exp);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ldexp calculates the value of $x * (2^{exp})$.

Return Value

ldexp returns the value of $x*(2^{exp})$. If an overflow results, the function returns +HUGE_VAL for a positive result or -HUGE_VAL for a negative result, and sets errno to ERANGE.

This example computes *y* as 1.5 times 2 to the fifth power ($1.5*2^5$):

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y;
   int  p;

    x = 1.5;
    p = 5;
    y = ldexp(x, p);
    printf("%lf times 2 to the power of %d is %lf\n", x, p, y);
   return  0;

   /*************************************************************************
        The output should be:

        1.500000 times 2 to the power of 5 is 48.000000
   *************************************************************************/
}
```

"exp — Calculate Exponential Function" on page 181
"frexp — Separate Floating-Point Value" on page 244
"modf — Separate Floating-Point Value" on page 412
"<math.h>" on page 770

ldiv — Perform Long Division

Format

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

Description

Language Level: ANSI, SAA, XPG4

ldiv calculates the quotient and remainder of the division of *numerator* by *denominator*.

Return Value

ldiv returns a structure of type ldiv_t, containing both the quotient (long int quot) and the remainder (long int rem). If the value cannot be represented, the return value is undefined. If *denominator* is 0, an exception is raised.

This example uses ldiv to calculate the quotients and remainders for a set of two dividends and two divisors.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long int num[2] =    { 45,-45 };
    long int den[2] =    { 7,-7 };
    ldiv_t ans;             /* ldiv_t is a struct type containing two long ints:
                                    'quot' stores quotient; 'rem' stores remainder    */
   short  i,j;

    printf("Results of long division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++) {
            ans = ldiv(num[i], den[j]);
            printf("Dividend: %6ld    Divisor: %6ld", num[i], den[j]);
            printf("    Quotient: %6ld    Remainder: %6ld\n", ans.quot, ans.rem);
        }
   return  0;

   /*************************************************************************
        The output should be:

        Results of long division:
        Dividend:   45   Divisor:     7   Quotient:     6   Remainder:     3
        Dividend:   45   Divisor:    -7   Quotient:    -6   Remainder:     3
        Dividend:  -45   Divisor:     7   Quotient:    -6   Remainder:    -3
        Dividend:  -45   Divisor:    -7   Quotient:     6   Remainder:    -3
   *************************************************************************/
}
```

"div — Calculate Quotient and Remainder" on page 143
"lldiv — Perform Long Long Division" on page 345
"<stdlib.h>" on page 775

lfind - lsearch — Find Key in Array

Format

```
#include <search.h>
char *lfind(char *search_key, char *base,
                    unsigned int *num, unsigned int *width,
                    int (*compare)(const void *key, const void *element));
char *lsearch(char *search_key, char *base,
                      unsigned int *num, unsigned int *width,
                      int (*compare)(const void *key, const void *element));
```

Description

Language Level: XPG4, Extension

lfind and lsearch perform a linear search for the value *search_key* in an array of *num* elements, each of *width* bytes in size. The argument *base* points to the base of the array to be searched. If lsearch does not find the *search_key*, it adds the *search_key* to the end of the array and increments *num* by one. If lfind does not find the *search_key*, it does not add the *search_key* to the array.

Unlike bsearch, lsearch and lfind do not require that you sort the array first. points to the base of the array to be searched.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. Both lfind and lsearch call this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values:

Value	Meaning
Nonzero	*key* and *element* are different.
0	*key* and *element* are identical.

Note: In earlier releases of VisualAge C++, lfind and lsearch began with an underscore (_lfind and _lsearch). Because they are defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _lfind and _lsearch to lfind and lsearch for you.

Return Value

If *search_key* is found, both lsearch and lfind return a pointer to that element of the array to which *base* points. If *search_key* is not found, lsearch returns a pointer to a newly added item at the end of the array, while lfind returns NULL.

This example uses lfind to search for the keyword PATH in the command-line arguments.

```
#include <search.h>
#include <string.h>
#include <stdio.h>

#define   CNT                 2

/* Note: Library always calls functions internally with _Optlink                    */
/*          linkage convention.    Ensure that compare() is always                    */
/*          _Optlink.                                                                  */

int _Optlink compare(const void *arg1,const void *arg2)
{
    return (strncmp(*(char **)arg1, *(char **)arg2, strlen(*(char **)arg1)));
}

int main(void)
{
   char  **result;
    char *key = "PATH";
    unsigned int num = CNT;
    char *string[CNT] =    {
        "PATH = d:\\david\\matthew\\heather\\ed\\simon","LIB = PATH\\abc" };

    /* The following statement finds the argument that starts with "PATH"          */

    if ((result = (char **)lfind((char *)&key, (char *)string, &num,
                        sizeof(char *), compare)) != NULL)
        printf("%s found\n", *result);
   else
        printf("PATH not found \n");
   return  0;

   /*************************************************************************
        The output should be:

        PATH = d:\david\matthew\heather\ed\simon found
   *************************************************************************/
}
```

"bsearch — Search Arrays" on page 71
"qsort — Sort Array" on page 446
"<search.h>" on page 772

llabs — Calculate Absolute Value of Long Long Integer

Format

```
#include <stdlib.h>
long long int llabs(long long int n);
```

Description

Language Level: Extension

llabs produces the absolute value of its long long integer argument *n*. The result may be undefined when the argument is equal to LONGLONG_MIN, the smallest available long long integer (-9 223 372 036 854 775 808). The value LONGLONG_MIN is defined in the <limits.h> include file.

Return Value

llabs returns the absolute value of *n*. There is no error return value.

This example computes *y* as the absolute value of the long long integer -41567.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long long x,y;

    x = -41567L;
    y = llabs(x);
    printf("The absolute value of %lld is %lld\n", x, y);
   return  0;

   /*****************************************************************************
       The output should be:

       The absolute value of -41567 is 41567
   *****************************************************************************/
}
```

"abs — Calculate Integer Absolute Value" on page 40
"fabs — Calculate Floating-Point Absolute Value" on page 182
"labs — Calculate Absolute Value of Long Integer" on page 339
"<limits.h>" on page 766

lldiv — Perform Long Long Division

Format

```
#include <stdlib.h>
lldiv_t lldiv(long long int numerator, long long int denominator);
```

Description

Language Level: Extension

lldiv calculates the quotient and remainder of the division of *numerator* by *denominator*.

Return Value

lldiv returns a structure of type lldiv_t, containing both the quotient (long long int quot) and the remainder (long long int rem). If the value cannot be represented, the return value is undefined. If *denominator* is 0, an exception is raised.

This example uses lldiv to calculate the quotients and remainders for a set of two dividends and two divisors.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long long int num[2] =    { 45,-45 };
    long long int den[2] =    { 7,-7 };
    lldiv_t ans;              /* lldiv_t is a struct type containing two long long ints:
                                    'quot' stores quotient; 'rem' stores remainder    */
   short  i,j;

    printf("Results of long long division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++) {
            ans = lldiv(num[i], den[j]);
           printf("Dividend:  %4lld      Divisor:  %4lld      Quotient:  %4lld      "
                    "Remainder: %4lld\n", num[i], den[j], ans.quot, ans.rem);
        }
   return  0;

   /*************************************************************************
        The output should be:

        Results of long long division:
        Dividend:    45    Divisor:      7    Quotient:      6    Remainder:      3
        Dividend:    45    Divisor:     -7    Quotient:     -6    Remainder:      3
        Dividend:   -45    Divisor:      7    Quotient:     -6    Remainder:     -3
        Dividend:   -45    Divisor:     -7    Quotient:      6    Remainder:     -3
   *************************************************************************/
}
```

"div — Calculate Quotient and Remainder" on page 143
"ldiv — Perform Long Division" on page 341
"<stdlib.h>" on page 775

_llrotl - _llrotr — Rotate Bits of Unsigned Long Long Integer

Format

```
#include <stdlib.h>      /* also in <builtin.h> */
unsigned long long _llrotl(unsigned long long value, int shift);
unsigned long long _llrotr(unsigned long long value, int shift);
```

Description

Language Level: Extension

These functions take a 8-byte unsigned (long long) integer *value* and rotate it by *shift* bits. _llrotl. rotates to the left, and _llrotr. to the right.

Note: Both _llrotl and _llrotr are built-in functions, which means they are implemented as inline instructions and have no backing code in the library. For this reason:

You cannot take the address of these functions.
You cannot parenthesize a call to either function. (Parentheses specify a call to the backing code for the function in the runtime library.)

Return Value

Both functions return the rotated value. There is no error return.

This example uses _llrotr and _llrotl with different shift values to rotate the integer value 0x01234567:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned long long val = 0x0123456789abcdefull;

    printf("The value of 0x%16.16llx rotated 4 bits to the left is 0x%16.16llx\n", val,
        _llrotl(val,  4));
    printf("The value of 0x%16.16llx rotated 16 bits to the right is 0x%16.16llx\n",
         val, _llrotr(val, 16));
   return  0;

   /*****************************************************************************
    The output should be:

    The value of 0x0123456789abcdef rotated 4 bits to the left is 0x123456789abcdef0
    The value of 0x0123456789abcdef rotated 16 bits to the right is 0xcdef0123456789ab
   *****************************************************************************/
}
```

"_crotl – _crotr — Rotate Bits of Character Value" on page 102
"_lrotl - _lrotr — Rotate Bits of Unsigned Long Value" on page 364
"_rotl - _rotr — Rotate Bits of Unsigned Integer" on page 483
"_srotl - _srotr — Rotate Bits of Unsigned Short Value" on page 529
"swab — Swap Adjacent Bytes" on page 608
"<builtin.h>" on page 761

“<stdlib.h>” on page 775

_loadmod — Load DLL

Format

#include <stdlib.h>
int _loadmod(char **module_name*, unsigned long **module_handle*);

Description

Language Level: Extension

_loadmod loads a dynamic link library (DLL) for the calling process. The *module_name* is the name of the DLL to be loaded, and the *module_handle* is the file handle associated with the DLL.

Note: _loadmod and _freemod perform exactly the same function as the Windows APIs LoadLibrary and FreeLibrary. They are provided for compatibility with earlier VisualAge for C++ releases only, and will not be supported in future versions. Use LoadLibrary and FreeLibrary instead. For more details on these APIs, see the *Win32 Programmer's Reference*.

If the operation is successful, _loadmod returns the handle of the loaded DLL to the calling process. The process can use this handle with the Win32 API GetProcAddress to get the address of the required function from within the DLL. Once the reference is established, the calling process can then call the specific function.

Return Value

_loadmod returns 0 if the DLL was successfully loaded. If unsuccessful, _loadmod returns -1, and sets **errno** to one of the following values:

Value	Meaning
EMODNAME	The specified *module_name* is not valid.
EOS2ERR	A system error occurred. Check _doserrno for the specific Windows error code.

In this example, _loadmod loads the DLL mark and gets the address of the function PLUS within the DLL. It then calls that function, which adds two integers passed to it. The types FARPROC and HMODULE are required to use the Win32 GetProcAddress API, and are defined by including the <windows.h> header file.

Note: To run this program, you must first create mark.dll. Copy the code for the PLUS function into a file called mark.c, and the code for the .DEF file into mark.def. Eliminate the comments from these two files. Compile with the command:

```
icc /Ge- mark.c mark.def
```

You can then run the example.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

typedef (_Optlink *fptr)(int, int);

int main(void) {

    int x = 4, y = 7;
    unsigned long handle;
    char *modname = "MARK";                          /* DLL name                 */
    char *fname = "PLUS";                           /* function name            */
    fptr faddr;                                     /* pointer to function    */

    /*    dynamically load the 'mark' DLL                                        */

    if (_loadmod(modname, &handle)) {
        printf("Error loading module %s\n", modname);
       return  EXIT_FAILURE;
    }

    /* get function address from DLL                 */

    faddr = (fptr)GetProcAddress((HMODULE)handle, fname);
    if (NULL != faddr) {
        printf("Calling the function from the %s DLL to add %d and %d\n",
                modname, x, y);
        printf("The result from the function call is %d\n", faddr(x, y));
    }
   else  {
        DWORD rc = GetLastError();
        printf("Error locating address of function %s, rc=%d\n", fname, rc);
       _freemod(handle);
       return  EXIT_FAILURE;
    }

    if (_freemod(handle)) {
        printf("Error in freeing the %s DLL module\n", modname);
       return  EXIT_FAILURE;
    }

    printf("Reference to the %s DLL module has been freed\n", modname);

   /*****************************************************************************
         The output should be:

         Calling the function from MARK DLL to add 4 and 7
         The result of the function call is 11
         Reference to the MARK DLL module has been freed
   *****************************************************************************/
}
```

"_freemod — Free User DLL" on page 240
"<stdlib.h>" on page 775

localdtconv — Return Date and Time Formatting Convention

Format

```
#include <locale.h>
struct dtconv *localdtconv(void);
```

Description

Language Level: Extension

localdtconv determines the date and time formatting conventions of the current locale and places the information in a structure of type struct dtconv. The dtconv structure is described in detail on page 768.

Subsequent calls to localdtconv or to setlocale with LC_ALL or LC_TIME can overwrite the structure.

Return Value

localdtconv returns the address of the dtconv structure.

This example sets the Spanish locale, calls localdtconv to determine how date and time are formatted, and prints the fields from the resulting structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <stdlib.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "es_es.ibm-437"        /* OS/2 name              */
#else
    #define LOCNAME "es_es.ibm-1252"       /* Windows name           */
#endif

int main(void)
{
    struct dtconv *p;
    int                 i;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
```

```
    } else {
        p = localdtconv();
        printf("SPAIN date/time convention.\n");
        printf(" Abbreviated month names:");
        for (i = 0; i < 12; ++i)
            printf(" %s", p->abbrev_month_names[i]);
        printf("\n Full month names:");
        for (i = 0; i < 12; ++i)
            printf(" %s", p->month_names[i]);
        printf("\n Abbreviated day names:");
        for (i = 0; i < 7; ++i)
            printf(" %s", p->abbrev_day_names[i]);
        printf("\n Full day names:");
        for (i = 0; i < 7; ++i)
            printf(" %s", p->day_names[i]);
        printf("\n Date and time format:    %s\n", p->date_time_format);
        printf(" Date format:    %s\n",                  p->date_format);
        printf(" Time format:    %s\n",                  p->time_format);
        printf(" AM string:    %s\n",                    p->am_string);
        printf(" PM string:    %s\n",                    p->pm_string);
        printf(" Long date format:    %s\n",            p->time_format_ampm);
    }
    return  0;

    /*****************************************************************************
        The output should be similar to :

        SPAIN date/time convention.
          Abbreviated month names: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
          Full month names: Enero Febrero Marzo Abril Mayo Junio Julio Agosto
                                Septiembre Octubre Noviembre Diciembre
          Abbreviated day names: Do Lu Ma Mi Ju Vi Sa
          Full day names: Domingo Lunes Martes Miercoles Jueves Viernes Sabado
          Date and time format:    %a %e %b %H:%M:%S %Y
         Date  format:    %d/%m/%y
         Time  format:    %H:%M:%S
         AM  string:
         PM  string:
          Long date format:    %I:%M:%S %p
    *****************************************************************************/
}
```

"localeconv — Retrieve Information from the Environment" on page 352
"localtime — Convert Time" on page 356
"setlocale — Set Locale" on page 499
"strftime — Convert to Formatted Time" on page 557
"strptime — Convert to Formatted Date and Time" on page 576
"wcsftime — Convert to Formatted Date and Time" on page 719
"<locale.h>" on page 766

localeconv — Retrieve Information from the Environment

Format

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

Language Level: ANSI, SAA, XPG4

localeconv retrieves information about the environment for the current locale and places the information in a structure of type struct lconv. Subsequent calls to localeconv, or to setlocale with the argument LC_ALL, LC_MONETARY, or LC_NUMERIC, can overwrite the structure.

The structure contains the members listed below. Defaults shown are for the "C" locale. Pointers to strings with a value of "" indicate that the value is not available in this locale or is of zero length. Character types with a value of UCHAR_MAX indicate that the value is not available in this locale.

Element	Purpose of Element	Default
char *decimal_point	Decimal-point character for formatting nonmonetary quantities.	"."
char *thousands_sep	Character used to separate groups of digits to the left of the decimal-point character in formatted nonmonetary quantities.	""
char *grouping	Size of each group of digits in formatted nonmonetary quantities. The value of each character in the string determines the number of digits in a group. CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *int_curr_symbol	International currency symbol. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) separates the international currency symbol from the monetary quantity.	""
char *currency_symbol	Local currency symbol.	""
char *mon_decimal_point	Decimal-point character for formatting monetary quantities.	"."
char *mon_thousands_sep	Separator for digits in formatted monetary quantities.	""

Element	Purpose of Element	Default
char *mon_grouping	Size of each group of digits in formatted monetary quantities. The value of each character in the string determines the number of digits in a group. CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *positive_sign	Positive sign used in monetary quantities.	""
char *negative_sign	Negative sign used in monetary quantities.	""
char int_frac_digits	Number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.	UCHAR_MAX
char frac_digits	Number of digits to the right of the decimal place in monetary quantities.	UCHAR_MAX
char p_cs_precedes	1 if the currency_symbol precedes the value for a nonnegative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char p_sep_by_space	1 if the currency_symbol is separated by a space from the value of a nonnegative formatted monetary quantity; 0 if it is not.	UCHAR_MAX
char n_cs_precedes	1 if the currency_symbol precedes the value for a negative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char n_sep_by_space	1 if the currency_symbol is separated by a space from the value of a negative formatted monetary quantity; 0 if it is not.	UCHAR_MAX
char p_sign_posn	Position of the positive_sign for a nonnegative formatted monetary quantity.	UCHAR_MAX
char n_sign_posn	Position of the negative_sign for a negative formatted monetary quantity.	UCHAR_MAX
char *left_parenthesis	Symbol to appear to the left of a negative-valued monetary symbol (such as a loss or deficit).	"("
char *right_parenthesis	Symbol to appear to the right of a negative-valued monetary symbol (such as a loss or deficit).	")"
char *debit_sign	String to indicate a non-negative-valued formatted monetary quantity.	"DB"
char *credit_sign	String to indicate a negative-valued formatted monetary quantity.	"CR"

The grouping and mon_grouping members can have the following values:

Value	Meaning
CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the rest of the digits.
other	The number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The n_sign_posn and p_sign_posn elements can have the following values:

Value	Meaning
0	The quantity and currency_symbol are enclosed in parentheses.
1	The sign precedes the quantity and currency_symbol.
2	The sign follows the quantity and currency_symbol.
3	The sign precedes the currency_symbol.
4	The sign follows the currency_symbol.
5	Use debit_sign for p_sign_posn or credit_sign for n_sign_posn.

Return Value

localeconv returns a pointer to the lconv structure.

This example prints out the default decimal point for your locale and then the decimal point for the LC_C_FRANCE locale.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "fr_fr.ibm-437"        /* OS/2 name             */
#else
    #define LOCNAME "fr_fr.ibm-1252"       /* Windows name          */
#endif

int main(void)
{
    struct lconv *mylocale;

    mylocale = localeconv();
    printf("Default decimal point is a %s\n", mylocale->decimal_point);
    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
    }
    mylocale = localeconv();
    printf("France's decimal point is a %s\n", mylocale->decimal_point);
   return  0;

   /*************************************************************************
        The output should be similar to :

        Default decimal point is a .
        France's decimal point is a ,
   *************************************************************************/
}
```

"setlocale — Set Locale" on page 499
"<locale.h>" on page 766

localtime — Convert Time

Format

```
#include <time.h>
struct tm *localtime(const time_t *timeval);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

localtime breaks down *timeval*, corrects for the local time zone and Daylight Saving Time, if appropriate, and stores the corrected time in a structure of type tm. See "gmtime — Convert Time" on page 289 for a description of the fields in a tm structure.

The time value is usually obtained by a call to the time function.

Notes:

1. gmtime and localtime may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
2. The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Return Value

localtime returns a pointer to the structure result. If unsuccessful, it returns NULL.

This example queries the system clock and displays the local time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
     struct tm *newtime;
    time_t  ltime;

    time(&ltime);
     newtime = localtime(&ltime);
     printf("The date and time is %s", asctime(newtime));
    return  0;

    /*****************************************************************************
         The output should be similar to:

         The date and time is Wed Oct 31 15:00:00 1995
    *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"mktime — Convert Local Time" on page 410
"time — Determine Current Time" on page 625
"<time.h>" on page 779

log — Calculate Natural Logarithm

Format

```
#include <math.h>
double log(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

log calculates the natural logarithm (base e) of *x*.

Return Value

log returns the computed value. If *x* is negative, log sets errno to EDOM and may return the value -HUGE_VAL. If *x* is zero, log returns the value -HUGE_VAL, and may set errno to ERANGE.

This example calculates the natural logarithm of 1000.0.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x = 1000.0,y;

    y = log(x);
    printf("The natural logarithm of %lf is %lf\n", x, y);
   return  0;

   /*************************************************************************
       The output should be:

       The natural logarithm of 1000.000000 is 6.907755
   *************************************************************************/
}
```

"exp — Calculate Exponential Function" on page 181
"log10 — Calculate Base 10 Logarithm" on page 359
"pow — Compute Power" on page 428
"<math.h>" on page 770

log10 — Calculate Base 10 Logarithm

Format

```
#include <math.h>
double log10(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

log10 calculates the base 10 logarithm of *x*.

Return Value

log10 returns the computed value. If *x* is negative, log10 sets errno to EDOM and may return the value -HUGE_VAL. If *x* is zero, log10 returns the value -HUGE_VAL, and may set errno to ERANGE.

This example calculates the base 10 logarithm of 1000.0.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x = 1000.0,y;

    y = log10(x);
    printf("The base 10 logarithm of %lf is %lf\n", x, y);
   return  0;

   /*************************************************************************
        The output should be:

        The base 10 logarithm of 1000.000000 is 3.000000
   *************************************************************************/
}
```

"exp — Calculate Exponential Function" on page 181
"log — Calculate Natural Logarithm" on page 358
"pow — Compute Power" on page 428
"<math.h>" on page 770

longjmp — Restore Stack Environment

Format

#include <setjmp.h>
void longjmp(jmp_buf *env*, int *value*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

longjmp restores a stack environment previously saved in *env* by setjmp. setjmp and longjmp provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to setjmp causes the current stack environment to be saved in *env*. A subsequent call to longjmp restores the saved environment and returns control to a point in the program corresponding to the setjmp call. Execution resumes as if the setjmp call had just returned the given *value*.

All variables (except register variables) that are accessible to the function that receives control contain the values they had when longjmp was called. The values of variables that are allocated to registers by the compiler are unpredictable. Because any auto variable could be allocated to a register in optimized code, you should consider the values of all auto variables to be unpredictable after a longjmp call.

Note: Ensure that the function that calls setjmp does not return before you call the corresponding longjmp function. Calling longjmp after the function calling setjmp returns causes unpredictable program behavior.

The *value* argument must be nonzero. If you give a zero argument for *value*, longjmp substitutes 1 in its place.

C++ Considerations: When you call setjmp in a C++ program, ensure that that part of the program does not also create C++ objects that need to be destroyed. When you call longjmp, objects existing at the time of the setjmp call will still exist, but any destructors called after setjmp are not called. For example, given the following program:

```
#include <stdio.h>
#include <setjmp.h>

class A {
    int  i;
    public:
         A(int I) {i = I; printf("Constructed at line %d\n", i);}
         ~A() {printf("Destroying object constructed at line %d\n",i);}
    };
jmp_buf jBuf;

int main(void) {
    A  a1(__LINE__);
     if (!setjmp(jBuf)) {
        A  a2(__LINE__);
         printf("Press y (and Enter) to longjmp; anything else to return norm
         char c = getchar();
         if (c == 'y')
             longjmp(jBuf,  1);
    }
    A  a3(__LINE__);
    return  0;
}
```

If you return normally, the output would be:

```
Constructed at line 13
Constructed at line 15
Press y (and Enter) to longjmp; anything else to return normally
x
Destroying object constructed at line 15
Constructed at line 21
Destroying object constructed at line 21
Destroying object constructed at line 13
```

If you called longjmp, the output would be:

```
Constructed at line 13
Constructed at line 15
Press y (and Enter) to longjmp; anything else to return normally
y
Constructed at line 21
Destroying object constructed at line 21
Destroying object constructed at line 13
```

Notice that the object from line 15 is not destroyed.

Return Value longjmp does not use the normal function call and return mechanisms; it has no return value.

This example saves the stack environment at the statement:

```
if(setjmp(mark) != 0) ...
```

When the system first performs the if statement, it saves the environment in mark and sets the condition to FALSE because setjmp returns a 0 when it saves the environment. The program prints the message:

```
setjmp has been called
```

The subsequent call to function p tests for a local error condition, which can cause it to call longjmp. Then, control returns to the original setjmp function using the environment saved in mark. This time, the condition is TRUE because -1 is the return value from longjmp. The example then performs the statements in the block, prints longjmp has been called, calls recover, and leaves the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int p(void);
int recover(void);
jmp_buf mark;

int main(void)
{
    if (setjmp(mark) != 0) {
        printf("longjmp has been called\n");
        recover();
        exit(1);
    }
    printf("setjmp has been called\n");

    p();

    return  0;

    /*****************************************************************************
        The output should be:

        setjmp has been called
        longjmp has been called
    *****************************************************************************/
}

int p(void)
{
    int error = 0;

    error = 9;

    if (error != 0)
        longjmp(mark,  -1);
    return  0;
}

int recover(void)
{
    return  0;
}
```

"setjmp — Preserve Environment" on page 497
goto in the *Language Reference*
"<setjmp.h>" on page 773

_lrotl - _lrotr — Rotate Bits of Unsigned Long Value

Format

```
#include <stdlib.h>      /* also in <builtin.h> */
unsigned long _lrotl(unsigned long value, int shift);
unsigned long _lrotr(unsigned long value, int shift);
```

Description

Language Level: Extension

_lrotl and _lrotr rotate the unsigned long integer *value* by *shift* bits. _lrotl rotates to the left, and _lrotr to the right.

Note: Both _lrotl and _lrotr are built-in functions, which means they are implemented as inline instructions and have no backing code in the library. For this reason:

You cannot take the address of these functions.
You cannot parenthesize a call to either function. (Parentheses specify a call to the function's backing code, and these functions have none.)

Return Value

Both functions return the rotated value. There is no error return value.

This example uses _lrotl and _lrotr with different shift values to rotate the long integer value 0x01234567.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned long val = 0X01234567;

    printf("The value of 0x%8.8lx rotated 4 bits to the left is 0x%8.8lx\n", val,
        _lrotl(val,  4));
    printf("The value of 0x%8.8lx rotated 16 bits to the right is 0x%8.8lx\n",
         val, _lrotr(val, 16));
   return  0;

   /*****************************************************************************
        The output should be:

        The value of 0x01234567 rotated 4 bits to the left is 0x12345670
        The value of 0x01234567 rotated 16 bits to the right is 0x45670123
   *****************************************************************************/
}
```

"_crotl – _crotr — Rotate Bits of Character Value" on page 102
"_rotl - _rotr — Rotate Bits of Unsigned Integer" on page 483
"_srotl - _srotr — Rotate Bits of Unsigned Short Value" on page 529
"<builtin.h>" on page 761
"<stdlib.h>" on page 775

lseek — Move File Pointer

Format

```
#include <io.h>    /* constants in <stdio.h> */
long lseek(int handle, long offset, int origin);
```

Description

Language Level: XPG4, Extension

lseek moves any file pointer associated with *handle* to a new location that is *offset* bytes from the *origin*. The next operation on the file takes place at the new location. The *origin* must be one of the following constants, defined in <stdio.h>:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file.

lseek can reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file; the data between the EOF and the new file position is undefined. (See "_chsize — Alter Length of File" on page 85 for more information on extending the length of the file.) An attempt to position the pointer before the beginning of the file causes an error.

Note: In earlier releases of VisualAge C++, lseek began with an underscore (_lseek). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _lseek to lseek for you.

Return Value

lseek returns the offset, in bytes, of the new position from the beginning of the file. A return value of -1L indicates an error, and lseek sets errno to one of the following values:

Value	Meaning
EBADF	The file handle is incorrect.
EINVAL	The value for *origin* is incorrect, or the position specified by *offset* is before the beginning of the file.
EOS2ERR	The call to the operating system was not successful.

On devices incapable of seeking (such as keyboards and printers), lseek returns -1 and sets errno to EOS2ERR.

This example opens the file sample.dat and, if successful, moves the file pointer to the eighth position in the file. The example then attempts to read bytes from the file, starting at the new pointer position, and reads them into the buffer.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
    long  length;
    int  fh;
    char  buffer[20];

     memset(buffer, '\0', 20);                    /* Initialize the buffer                */
    printf("\nCreating  sample.dat.\n");
     system("echo Sample Program > sample.dat");
     if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
          perror("Unable to open sample.dat.");
          exit(EXIT_FAILURE);
    }
     if (-1 == lseek(fh, 7, SEEK_SET)) {          /* place the file pointer at the     */
          perror("Unable to lseek");               /* eighth position in the file       */
          close(fh);
          return  EXIT_FAILURE;
    }
     if (8 != read(fh, buffer, 8)) {
          perror("Unable to read from sample.dat.");
          close(fh);
          return  EXIT_FAILURE;
    }
     printf("Successfully read in the following:\n%s.\n", buffer);
    close(fh);
    return  0;

    /*************************************************************************
          The output should be:

         Creating  sample.dat.
          Successfully read in the following:
          Program  .
    *************************************************************************/
}
```

"_chsize — Alter Length of File" on page 85
"fseek — Reposition File Position" on page 247
"_tell — Get Pointer Position" on page 619
"<io.h>" on page 764
"<stdio.h>" on page 774

_ltoa — Convert Long Integer to String

Format

#include <stdlib.h>
char *_ltoa(long *value*, char **string*, int *radix*);

Description

Language Level: Extension

_ltoa converts the digits of the given long integer *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

Note: The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes, including the null character (\0).

Return Value

_ltoa returns a pointer to *string*. There is no error return value.

This example converts the long integer -255L to a decimal, binary, and hex value, and stores its character representation in the array buffer.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  buffer[35];
   char  *p;

    p = _ltoa(-255L, buffer, 10);
    printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
    p = _ltoa(-255L, buffer, 2);
    printf("The result of _ltoa(-255) with radix of 2\n       is %s\n", p);
    p = _ltoa(-255L, buffer, 16);
    printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
   return  0;

  /*****************************************************************************
       The output should be:

       The result of _ltoa(-255) with radix of 10 is -255
       The result of _ltoa(-255) with radix of 2
            is  11111111111111111111111100000001
       The result of _ltoa(-255) with radix of 16 is ffffff01
  *****************************************************************************/
}
```

"atol — Convert Character String to Long Integer" on page 60
"_ecvt — Convert Floating-Point to Character" on page 166
"_fcvt — Convert Floating-Point to String" on page 192
"_gcvt — Convert Floating-Point to String" on page 268
"_itoa — Convert Integer to String" on page 336
"strtol — Convert Character String to Long Integer" on page 594

_ltoa

"_ultoa — Convert Unsigned Long Integer to String" on page 675
"wcstol — Convert Wide-Character to Long Integer" on page 741
"<stdlib.h>" on page 775

__lxchg — Exchange Integer Value with Memory

Format

```
#include <builtin.h>
int _Builtin __lxchg(volatile int*lockptr, int value);
```

Description

Language Level: Extension

__lxchg puts the specified *value* in the memory location pointed to by *lockptr*, and returns the value that was previously in that location.

Use this function to implement fast-RAM semaphores to serialize access to a critical resource (so that only one thread can use it at a time). You can also use system semaphores to serialize resource access, but they are slower. Typically you would create both a fast-RAM semaphore and a system semaphore for the resource.

To implement a fast-RAM semaphore, allocate a volatile memory location *lockptr* (for __lxchg, it must be of type int), and statically initialize it to 0 to indicate that the resource is free (not being used). To give a thread access to the resource, call __lxchg from the thread to exchange a non-zero value with that memory location. If __lxchg returns 0, the thread can access the resource; it has also set the memory location so that other threads can tell the resource is in use. When your thread no longer needs the resource, call __lxchg again to reset the memory location to 0.

If __lxchg returns a non-zero value, another thread is already using the resource, and the calling thread must wait for access using a Windows semaphore. You could then use the Windows semaphore to inform your waiting thread when the resource is unlocked by the thread currently using it.

It is important that you set the memory to a nonzero value when you are using the resource. You can use the same nonzero value for all threads, or a unique value for each.

Note: __lxchg is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of __lxchg.
You cannot parenthesize a call to __lxchg. (Parentheses specify a call to the function's backing code, and __lxchg has none.)

Return Value

__lxchg returns the previous value stored in the memory location pointed to by *lockptr*.

__lxchg

This example creates five separate threads, each associated with a State. At most two threads can have a State of Eating at the same time. When a thread calls PickUpChopSticks, that function checks the states of the preceding threads to make sure they are not already Eating. If the call to __lxchg is successful, the thread has locked the Lock semaphore and can change its State to Eating. It then calls __lxchg a second time to unlock the semaphore, and returns.

If __lxchg cannot lock the semaphore, another thread has it locked. The current thread then suspends itself briefly with Sleep and tries again. The semaphore is used to ensure that two threads cannot simultaneously change their State to Eating, which would cause a deadlock. (Note that although the semaphore solves the problem of deadlock, it is not an optimal solution; some threads may never get the semaphore or the State of Eating.)

Note: You must compile this example with the multithread (/Gm) option. The example runs in an infinite loop; press Ctrl-C to end it.

```
   #pragma strings(readonly)

#if   (1 == __TOS_OS2__)
       #define     INCL_DOS                                  /*   For OS/2   */
       #include <os2.h>
       #define SLEEP   DosSleep(1)
#else
      #include <windows.h>                                 /*   For Windows   */
      #define SLEEP   Sleep(1)
#endif

   #include <stdlib.h>
   #include <stdio.h>
   #include <builtin.h>

    typedef enum {
       Thinking,
       Eating,
       Hungry
   } States;
```

```
#define UNLOCKED 0
#define  LOCKED     1
#define NUM_PHILOSOPHERS 5

static volatile int Lock;
static States State[NUM_PHILOSOPHERS];
static const int NameMap[NUM_PHILOSOPHERS] = {0, 1, 2, 3, 4};
static const char * const Names[NUM_PHILOSOPHERS] = {"Plato",
                                                     "Socrates",
                                                     "Kant",
                                                     "Hegel",
                                                     "Nietsche"};

void PickUpChopSticks(int Ident);
void PutDownChopSticks(int Ident);
void Think(int Ident);
void Eat(int Ident);
void _Optlink StartPhilosopher(void *pIdent);

void _Optlink StartPhilosopher(void *pIdent)
{
     int Ident = *(int *)pIdent;
     while (1) {
          State[Ident] = Hungry;
          printf("%s hungry.\n", Names[Ident]);
          PickUpChopSticks(Ident);
          Eat(Ident);
          PutDownChopSticks(Ident);
          Think(Ident);
     }
}

int main(int argc, char *argv[], char *envp[])
{
    int                  i;
     unsigned long tid;

     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
          tid = _beginthread(StartPhilosopher, NULL, 8192, (void *)&NameMap[i]);
          if (tid == -1) {
               printf("Unable to start %s.\n", Names[i]);
          }
          else  {
               printf("%s started with Thread Id = %d.\n", Names[i], tid);
          }
     }
```

__lxchg

```
#if    (1 == __TOS_OS2__)
        DosWaitThread(&tid,  DCWW_WAIT);
#else
        WaitForSingleObject((HANDLE)tid,  INFINITE);
#endif
        return  0;
    }

    void PickUpChopSticks(int Ident)
    {
         while (1) {
              if (State[(Ident + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] != Eating &&
                    State[(Ident + 1) % NUM_PHILOSOPHERS] != Eating                    &&
                    !__lxchg(&Lock, LOCKED)) {
                       State[Ident] = Eating;
                      __lxchg(&Lock,  UNLOCKED);
                      return;
              }
              else  {
                   SLEEP;
              }
         }
    }

    void PutDownChopSticks(int Ident)
    {
         State[Ident] = Thinking;
    }

    void Eat(int Ident)
    {
         printf("%s eating.\n", Names[Ident]);
        SLEEP;
    }

    void Think(int Ident)
    {
         printf("%s thinking.\n", Names[Ident]);
        SLEEP;
    }
```

```
/*****************************************************************************
        The output should be similar to :

        Plato started with Thread Id = 2.
        Socrates started with Thread Id = 3.
        Kant started with Thread Id = 4.
        Hegel started with Thread Id = 5.
        Nietsche started with Thread Id = 6.
       Plato  hungry.
       Plato  eating.
       Socrates  hungry.
       Kant  hungry.
       Kant  eating.
       Hegel  hungry.
       Nietsche  hungry.
       Kant  thinking.
       Plato  thinking.
       Plato  hungry.
       Plato  eating.
       Kant  hungry.
       Kant  eating.
           :

*****************************************************************************/
```

"__cxchg — Exchange Character Value with Memory" on page 119
"__sxchg — Exchange Integer Value with Memory" on page 615
"<builtin.h>" on page 761

_makepath — Create Path

Format

#include <stdlib.h>
void _makepath(char *_path_, char *_drive_, char *_dir_, char *_fname_, char *_ext_);

Description

Language Level: Extension

_makepath creates a single path name, composed of a drive letter, directory path, file name, and file name extension.

The _path_ argument should point to an empty buffer large enough to hold the complete path name. The constant _MAX_PATH, defined in <stdlib.h>, specifies the maximum size allowed for _path_. The other arguments point to the following buffers containing the path name elements:

drive A letter (A, B, ...) corresponding to the desired drive and an optional following colon. _makepath inserts the colon automatically in the composite path name if it is missing. If _drive_ is a null character or an empty string, no drive letter or colon appears in the composite _path_ string.

dir The path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either slash (/) or backslash (\) or both can be used in a single _dir_ argument. If a trailing slash or backslash is not specified, it is inserted automatically. If _dir_ is a null character or an empty string, no slash is inserted in the composite _path_ string.

fname The base file name without any extensions.

ext The actual file name extension, with or without a leading period. _makepath inserts the period automatically if it does not appear in _ext_. If _ext_ is a null character or an empty string, no period is inserted in the composite _path_ string.

The size limits on the above four fields are those specified by the constants _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT, which are defined in <stdlib.h>. The composite path should be no larger than the _MAX_PATH constant also defined in <stdlib.h>; otherwise, the operating system does not handle it correctly.

Note: No checking is done to see if the syntax of the file name is correct.

Return Value

There is no return value.

This example builds a file name path from the specified components.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  path_buffer[_MAX_PATH];
   char  drive[_MAX_DRIVE];
   char  dir[_MAX_DIR];
   char  fname[_MAX_FNAME];
   char  ext[_MAX_EXT];

    _makepath(path_buffer, "c", "qc\\bob\\eclibref\\e", "makepath", "c");
    printf("Path created with _makepath: %s\n\n", path_buffer);
    _splitpath(path_buffer, drive, dir, fname, ext);
    printf("Path extracted with _splitpath:\n");
    printf("drive: %s\n", drive);
    printf("directory: %s\n", dir);
    printf("file name: %s\n", fname);
    printf("extension: %s\n", ext);
   return  0;

   /*************************************************************************
        The output should be:

        Path created with _makepath: c:qc\bob\eclibref\e\makepath.c

        Path extracted with _splitpath:
       drive:  c:
       directory:  qc\bob\eclibref\e\
        file name: makepath
       extension:  .c
   *************************************************************************/
}
```

"_fullpath — Get Full Path Name of Partial Path" on page 261
"_splitpath — Decompose Path Name" on page 523
"<stdlib.h>" on page 775

malloc — Reserve Storage Block

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void *malloc(size_t size);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

malloc reserves a block of storage of *size* bytes. Unlike calloc, malloc does not initialize all elements to 0.

Heap-specific, and debug versions of this function (_umalloc, and _debug_malloc) are also available. malloc always operates on the default heap. For more information about memory management, see Managing Memory in the *Programming Guide*.

Return Value

malloc returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. The return value is NULL if not enough storage is available, or if *size* was specified as zero.

This example prompts for the number of array entries you want and then reserves enough space in storage for the entries. If malloc was successful, the example assigns values to the entries and prints out each entry; otherwise, it prints out an error.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long *array;                                    /* start of the array                  */
    long *index;                                                    /* index variable     */
    int i;                                                          /* index variable     */
    int num;                                        /* number of entries of the array     */

    printf("Enter the size of the array\n");
   scanf("%i",  &num);

    /* allocate num entries                                                                */

    if ((index = array = (long *)malloc(num *sizeof(long))) != NULL) {
        for (i = 0; i < num; ++i)                   /* put values in array                 */
            *index++ = i;                                   /* using pointer notation          */
        for (i = 0; i < num; ++i)                   /* print the array out                 */
            printf("array[ %i ] = %i\n", i, array[i]);
   }
    else {                                                          /* malloc error       */
        perror("Out of storage");
       abort();
   }
   return  0;

   /*************************************************************************
        The output should be similar to:

        Enter the size of the array
       5
        array[ 0 ] = 0
        array[ 1 ] = 1
        array[ 2 ] = 2
        array[ 3 ] = 3
        array[ 4 ] = 4
   *************************************************************************/
}
```

"calloc — Reserve and Initialize Storage" on page 75
"_debug_malloc — Allocate Memory" on page 130
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"free — Release Storage Blocks" on page 238
"_mheap — Query Memory Heap for Allocated Object" on page 404
"_msize — Return Number of Bytes Allocated" on page 413
"realloc — Change Reserved Storage Block Size" on page 452
"_umalloc — Reserve Memory Blocks from User Heap" on page 677
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

_matherr — Process Math Library Errors

Format

#include <math.h>
int _matherr(struct exception *x);

Description

Language Level: Extension

_matherr processes errors generated by the functions in the math library. The math functions call _matherr whenever they detect an error. The _matherr function supplied with the VisualAge for C++ library performs no error handling and returns 0 to the calling function.

You can provide a different definition of _matherr to carry out special error handling. Be sure to use the /NOE link option, if you are using your own _matherr function. For VisualAge for C++ compiler, you can use the /B option on the icc command line to pass the /NOE option to the linker, as in the following:

```
icc /B"/NOE" yourfile.c
```

When an error occurs in a math routine, _matherr is called with a pointer to the following structure (defined in <math.h>) as an argument:

```
struct exception {
      int  type;
      char  *name;
       double arg1, arg2, retval;
};
```

The type field specifies the type of math error. It has one of the following values, defined in <math.h>:

Value	Meaning
DOMAIN	Argument domain error
OVERFLOW	Overflow range error
UNDERFLOW	Underflow range error
TLOSS	Total loss of significance
PLOSS	Partial loss of significance
SING	Argument singularity.

PLOSS is provided for compatibility with other compilers; VisualAge for C++ does not generate this value.

The name is a pointer to a null-ended string containing the name of the function that caused the error. The arg1 and arg2 fields specify the argument values that caused the error. If only one argument is given, it is stored in arg1. The retval is the default return value; you can change it.

Return Value

The return value from _matherr must specify whether or not an error actually occurred. If _matherr returns 0, an error message appears, and **errno** is set to an appropriate error value. If _matherr returns a nonzero value, no error message appears and **errno** remains unchanged.

This example provides a _matherr function to handle errors from the log or log10 functions. The arguments to these logarithmic functions must be positive double values. _matherr processes a negative value in an argument (a domain error) by returning the log of its absolute value. It suppresses the error message normally displayed when this error occurs. If the error is a zero argument or if some other routine produced the error, the example takes the default actions.

Note: You must compile this example with the /B"/NOE" compiler option.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int  value;

    printf("Trying to evaluate log10(-1000.00) will create a math exception.\n");
    value = log10(-1000.00);
    printf("The _matherr() exception handler evaluates the expression to\n");
    printf("log10(-1000.00) = %d\n", value);
   return  0;

   /*****************************************************************************
        The output should be:

        Trying to evaluate log10(-1000.00) will create a math exception.
       inside  _matherr
        The _matherr() exception handler evaluates the expression to
        log10(-1000.00) = 3
   *****************************************************************************/
}
```

_matherr

```
int _matherr(struct exception *x)
{
    printf("inside  _matherr\n");
     if (DOMAIN == x->type) {
          if (0 == strcmp(x->name, "log")) {
               x->retval = log(-(x->arg1));
              return  EXIT_FAILURE;
          }
          else
               if (0 == strcmp(x->name, "log10")) {
                    x->retval = log10(-(x->arg1));
                   return  EXIT_FAILURE;
               }
    }
     return 0;                                          /* Use default actions                */
}
```

"log — Calculate Natural Logarithm" on page 358
"log10 — Calculate Base 10 Logarithm" on page 359
"<math.h>" on page 770

max — Return Larger of Two Values

Format

#include <stdlib.h>
type max(*type a*, *type b*);

Description

Language Level: Extension

max compares two values and determines the larger of the two. The data *type* can be any arithmetic data type, signed or unsigned. Both arguments must have the same type for each call to max.

Note: Because max is a macro, if the evaluation of the arguments contains side effects (post-increment operators, for example), the results of both the side effects and the macro will be undefined.

Return Value

max returns the larger of the two values.

This example prints the larger of the two values, a and b.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 21;

    printf("The larger of %d and %d is %d\n", a, b, max(a, b));
   return  0;

  /*****************************************************************************
       The output should be:

       The larger of 10 and 21 is 21
  *****************************************************************************/
}
```

"min — Return Lesser of Two Values" on page 406
"<stdlib.h>" on page 775

mblen — Determine Length of Multibyte Character

Format

#include <stdlib.h>
int mblen(const char *_string_, size_t _n_);

Description

Language Level: ANSI, SAA, XPG4

mblen determines the length in bytes of the multibyte character pointed to by *string*. A maximum of *n* bytes is examined.

The behavior of mblen is affected by the LC_CTYPE category of the current locale.

Return Value

If *string* is NULL, mblen returns 0.

Note: On platforms that support shift states, mblen can also return a nonzero value to indicate that the multibyte encoding is state-dependent. Because VisualAge for C++ does not support state-dependent encoding, mblen always returns 0 when *string* is NULL.

If *string* is not NULL, mblen returns:

0 if *string* points to the null character
The number of bytes comprising the multibyte character
The value –1 if *string* does not point to a valid multibyte character.

This example uses mblen and mbtowc to convert a multibyte character into a single wide character.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    char         mb_string[] = "\x81\x41\x81\xc2" "b";
   int          length;
   wchar_t    widechar;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    length = mblen(mb_string, MB_CUR_MAX);
    mbtowc(&widechar, mb_string, length);
    printf("The wide character %lc has length of %d.\n", widechar, length);
   return  0;

   /*************************************************************************
        The output should be similar to :

        The wide character  A has length of 2.
   *************************************************************************/
}
```

“mbrlen — Calculate Length of Multibyte Character” on page 384
“mbrtowc — Convert Multibyte Character to Wide Character” on page 386
“mbsrtowcs — Convert Multibyte String to Wide-Character String” on page 389
“mbstowcs — Convert Multibyte String to Wide-Character String” on page 391
“mbtowc — Convert Multibyte Character to Wide Character” on page 393
“setlocale — Set Locale” on page 499
“strlen — Determine String Length” on page 564
“wcrtomb — Convert Wide Character to Multibyte Character” on page 706
“wcslen — Calculate Length of Wide-Character String” on page 722
“wcsrtombs — Convert Wide-Character String to Multibyte String” on page 733
“wctomb — Convert Wide Character to Multibyte Character” on page 753
“<locale.h>” on page 766
“<stdlib.h>” on page 775

mbrlen — Calculate Length of Multibyte Character

Format

```
#include <wchar.h>
size_t mbrlen(const char *string, size_t n,
              mbstate_t  *state);
```

Description

Language Level: ANSI 93

mbrlen is a restartable version of mblen and performs the same function. It determines the length in bytes of the multibyte character pointed to by *string*. A maximum of *n* bytes are examined.

With mbrlen, you can begin processing a string, and then another. Once you have begun processing the second string, you may switch back and continue with the first string. On platforms that support shift states, *state* represents the initial shift state of the string (0). If you read in only part of the string, mbrlen sets *state* to the string's shift state at the point you stopped. You can then call mbrlen again for that string and pass in the updated *state* value to continue reading where you left off.

Note: Because the Windows code pages do not have shift states, the *state* parameter is provided only for compatibility with other ANSI/ISO platforms. VisualAge for C++ ignores the value passed for *state*.

The behavior of mbrlen is affected by the LC_CTYPE category of the current locale.

Return Value

If *string* is a null pointer, mbrlen returns 0.

If *string* is not a null pointer, mbrlen returns the first of the following that applies:

0 If the next *n* or fewer bytes complete the valid multibyte character that corresponds to the null wide character.

positive
If the next *n* or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.

−2 If the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed.

−1 If an encoding error occurs (when the next *n* or fewer bytes do not form a complete and valid multibyte character). mbrlen sets errno to EILSEQ.

This example uses mbrlen to determine the length of the strings mbs1 and mbs2.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    char        mbs1[] = "abc";
    char        mbs2[] = "\x81\x41" "a" "\x81\xc1";
    mbstate_t   ss1 = 0;
    mbstate_t   ss2 = 0;
   size_t       length1,  length2;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
    }
    length1 =    mbrlen(mbs1, MB_CUR_MAX, &ss1);                  /* first char    */
    length2 =    mbrlen(mbs2, MB_CUR_MAX, &ss2);                  /* first char    */
    length1 += mbrlen(mbs1 + length1, MB_CUR_MAX, &ss1);     /* second char */
    length2 += mbrlen(mbs2 + length2, MB_CUR_MAX, &ss2);     /* second char */
    length1 += mbrlen(mbs1 + length1, MB_CUR_MAX, &ss1);     /* third char    */
    length2 += mbrlen(mbs2 + length2, MB_CUR_MAX, &ss2);     /* third char    */
    printf("Length of the first multibyte string = %d\n", length1);
    printf("Length of the second multibyte string = %d\n", length2);
   return  0;

   /*************************************************************************
        The output should be similiar to :

        Length of the first multibyte string = 3
        Length of the second multibyte string = 5
   *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbtowc — Convert Multibyte Character to Wide Character" on page 393
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbsrtowcs — Convert Multibyte String to Wide-Character String" on page 389
"setlocale — Set Locale" on page 499
"wcrtomb — Convert Wide Character to Multibyte Character" on page 706
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"<locale.h>" on page 766
"<wchar.h>" on page 780

mbrtowc — Convert Multibyte Character to Wide Character

Format

```
#include <wchar.h>
size_t mbrtowc (wchar_t *pwc, const char *string,
                size_t n, mbstate_t *ps);
```

Description

Language Level: ANSI 93

mbrtowc is a restartable version of mbtowc, and performs the same function. It first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to the corresponding wide character, and stores the converted character in the location pointed to by *pwc*, if *pwc* is not a null pointer. A maximum of *n* bytes are examined.

With mbrtowc, you can switch from one multibyte string to another. On systems that support shift states, *ps* represents the initial shift state of the string (0). If you read in only part of the string, mbrtowc sets *ps* to the string's shift state at the point you stopped. You can then call mbrtowc again for that string and pass in the updated *ps* value to continue reading where you left off.

Note: Because Windows code pages do not support shift states, the *ps* parameter is provided only for compatibility with other ANSI/ISO platforms. VisualAge for C++ ignores the value passed for *ps*.

The behavior of mbrtowc is affected by the LC_CTYPE category of the current locale.

Return Value

If *string* is a null pointer, mbrtowc returns 0.

If *string* is not a null pointer, mbrtowc returns the first of the following that applies:

0
: If the next *n* or fewer bytes complete the valid multibyte character that corresponds to the null wide character.

positive
: If the next *n* or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.

−2
: If the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed.

−1
: If an encoding error occurs (when the next *n* or fewer bytes do not form a complete and valid multibyte character). mbrtowc sets errno to EILSEQ.

This example uses mbrlen to move to the second character in a string, then calls mbrtowc to convert the multibyte character to a wide character.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    char            mbs1[] = "abc";
    char            mbs2[] = "\x81\x41\x81\x42" "m";
    mbstate_t   ss1 = 0;
    mbstate_t   ss2 = 0;
   size_t        length1,  length2;
   wchar_t       wc1,  wc2;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    length1 = mbrlen(mbs1, MB_CUR_MAX, &ss1);
    length2 = mbrlen(mbs2, MB_CUR_MAX, &ss2);
    mbrtowc(&wc1, mbs1 + length1, MB_CUR_MAX, &ss1);
    mbrtowc(&wc2, mbs2 + length2, MB_CUR_MAX, &ss2);
    printf("The second wide character in mbs1 is: <%lc>\n", wc1);
    printf("The second wide character in mbs2 is: <%lc>\n", wc2);
   return  0;

   /*************************************************************************
        The output should be similiar to :

        The second wide character in mbs1 is: <b>
        The second wide character in mbs2 is: < B>
   *************************************************************************/
}
```

“mblen — Determine Length of Multibyte Character” on page 382
“mbrlen — Calculate Length of Multibyte Character” on page 384
“mbsrtowcs — Convert Multibyte String to Wide-Character String” on page 389
“setlocale — Set Locale” on page 499
“wcrtomb — Convert Wide Character to Multibyte Character” on page 706
“wcsrtombs — Convert Wide-Character String to Multibyte String” on page 733
“<locale.h>” on page 766
“<wchar.h>” on page 780

mbsinit — Test Object for Initial State

Format

```
#include <wchar.h>
int mbsinit(mbstate_t *state);
```

Description

Language Level: Extension

mbsinit determines whether the *state* describes an initial conversion state.

Note: mbsinit is provided only for compatibility with EBCDIC systems that support conversion or shift states. The Windows code pages do not use shift states for multibyte character encoding.

Return Value

If *state* is a null pointer or points to 0, mbsinit returns a nonzero value. If *state* points to any other value, mbsinit returns 0. On systems that support shift states, mbsinit returns nonzero if *state* describes an initial conversion state, or 0 otherwise.

This example checks the conversion state to see if it is in the initial state.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char          *string = "ABC";
    mbstate_t state = 0;
   wchar_t     wc;

    mbrtowc(&wc, string, MB_CUR_MAX, &state);
    if (0 != mbsinit(&state))
        printf("In initial conversion state.\n");
   return  0;

   /*************************************************************************
        The output should be similar to :

        In initial conversion state.
   *************************************************************************/
}
```

"mbrlen — Calculate Length of Multibyte Character" on page 384
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbsrtowcs — Convert Multibyte String to Wide-Character String" on page 389
"setlocale — Set Locale" on page 499
"wcrtomb — Convert Wide Character to Multibyte Character" on page 706
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"<locale.h>" on page 766
"<wchar.h>" on page 780

mbsrtowcs — Convert Multibyte String to Wide-Character String

Format

```
#include <wchar.h>
size_t mbsrtowcs (wchar_t *dest, const char **src,
                  size_t len, mbstate_t *ps);
```

Description

Language Level: ANSI 93

mbsrtowcs is a restartable version of mbstowcs, and performs the same function. It converts the sequence of multibyte characters from the array indirectly pointed to by *src* into a sequence of corresponding wide characters, and then stores the converted characters in the array pointed to by *dest*.

Conversion continues up to and including the terminating wchar_t null character. The terminating null wide character is also stored. Conversion stops earlier if a sequence of bytes does not form a valid multibyte character, or when *len* codes have been stored into the array pointed to by *dest*. Each conversion takes place as if by a call to the mbrtowc function.

mbsrtowcs assigns the object pointed to by *src* either a null pointer (if conversion stopped because a terminating null character was reached) or the address just past the last multibyte character converted.

With mbsrtowcs, you can switch from one multibyte string to another. On platforms that support shift states, *ps* represents the initial shift state of the string (0). If you read in only part of the string, mbsrtowcs sets *ps* to the string's shift state at the point you stopped. You can then call mbsrtowcs again for that string and pass in the updated *ps* value to continue reading where you left off.

Note: Because Windows does not have shift states, the *ps* parameter is provided only for compatibility with other ANSI/ISO platforms. VisualAge for C++ ignores the value passed for *ps*.

The behavior of mbsrtowcs is affected by the LC_CTYPE category of the current locale.

Return Value

mbsrtowcs returns the number of multibyte characters successfully converted, not including the terminating null character. If *dest* is a null pointer, the value of *len* is ignored and mbsrtowcs returns the number of elements required for the converted wide characters.

If the input string contains an invalid multibyte character, mbsrtowcs sets errno to EILSEQ and returns (size_t)−1.

mbsrtowcs

This example uses mbsrtowcs to convert the multibyte characters in the arrays mbs1 and mbs2, and store them in the arrays wcs1 and wcs2.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define SIZE 10

int main(void)
{
    char              mbs1[] = "abc";
    char              mbs2[] = "\x81\x41" "m" "\x81\x42";
    const char    *pmbs1 = mbs1;
    const char    *pmbs2 = mbs2;
   mbstate_t    ss1  =  0;
   mbstate_t    ss2  =  0;
   wchar_t        wcs1[SIZE],  wcs2[SIZE];

    if (NULL == setlocale(LC_ALL, "ja_jp.ibm-932")) {
        printf("setlocale  failed.\n");
        exit(EXIT_FAILURE);
   }
    mbsrtowcs(wcs1, &pmbs1, SIZE, &ss1);
    mbsrtowcs(wcs2, &pmbs2, SIZE, &ss2);
    printf("The first wide character string is %ls.\n", wcs1);
    printf("The second wide character string is %ls.\n", wcs2);
   return  0;

   /*************************************************************************
        The output should be similiar to :

        The first wide character string is abc.
        The second wide character string is  Am B.
   *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbrlen — Calculate Length of Multibyte Character" on page 384
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbstowcs — Convert Multibyte String to Wide-Character String" on page 391
"setlocale — Set Locale" on page 499
"wcrtomb — Convert Wide Character to Multibyte Character" on page 706
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"<locale.h>" on page 766
"<wchar.h>" on page 780

mbstowcs — Convert Multibyte String to Wide-Character String

Format

#include <stdlib.h>
size_t mbstowcs(wchar_t *dest, const char *string, size_t len);

Description

Language Level: ANSI, SAA, XPG4

mbstowcs converts the multibyte character string pointed to by *string* into the wide-character array pointed to by *dest*. Depending on the encoding scheme used by the code set, the multibyte character string can contain any combination of single-byte or double-byte characters.

The conversion stops after *len* bytes in *dest* are filled or after a wchar_t null character is encountered. The terminating null character is converted to a wide character with the value 0; characters that follow it are not processed.

The behavior of mbstowcs is affected by the LC_CTYPE category of the current locale.

Return Value

If successful, mbstowcs returns the number of characters converted and stored in *dest*, not counting the terminating null character. The string pointed to by *dest* ends with a null character unless mbstowcs returns the value *len*.

If it encounters an invalid multibyte character, mbstowcs returns (size_t)–1. If *dest* is a null pointer, the value of *len* is ignored and mbstowcs returns the number of elements required for the converted wide characters.

This example uses mbstowcs to convert the multibyte character mbs to a a wide character string and store it in wcs.

mbstowcs

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define SIZE 10
#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    char              mbs[] = "\x81\x41" "m" "\x81\x42";
   wchar_t         wcs[SIZE];

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    mbstowcs(wcs, mbs, SIZE);
    printf("The wide character string is %ls.\n", wcs);
   return  0;

   /**************************************************************************
        The output should be similiar to :

        The wide character string is  Am B.
   **************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbsrtowcs — Convert Multibyte String to Wide-Character String" on page 389
"mbtowc — Convert Multibyte Character to Wide Character" on page 393
"setlocale — Set Locale" on page 499
"wcslen — Calculate Length of Wide-Character String" on page 722
"wcstombs — Convert Wide-Character String to Multibyte String" on page 743
"<locale.h>" on page 766
"<stdlib.h>" on page 775

mbtowc — Convert Multibyte Character to Wide Character

Format

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

Description

Language Level: ANSI, SAA, XPG4

mbtowc first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to the corresponding wide character, and stores it in the location pointed to by *pwc*, if *pwc* is not a null pointer. mbtowc examines a maximum of *n* bytes from *string*.

If *pwc* is a null pointer, the multibyte character is not converted.

The behavior of mbtowc is affected by the LC_CTYPE category of the current locale.

Return Value

If *string* is NULL, mbtowc returns 0.

Note: On platforms that support shift states, mbtowc can also return a nonzero value to indicate that the multibyte encoding is state dependent. Because VisualAge for C++ does not support state-dependent encoding, mbtowc always returns 0 when *string* is NULL.

If *string* is not NULL, mbtowc returns:

The number of bytes comprising the converted multibyte character, if *n* or fewer bytes form a valid multibyte character.
0 if *string* points to the null character.
–1 if *string* does not point to a valid multibyte character, and the next *n* bytes do not form a valid multibyte character.

This example uses mbtowc to convert the second multibyte character in mbs to a wide character.

mbtowc

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    char        mb_string[] = "\x81\x41\x81\x42" "c" "\x00";
   int          length;
   wchar_t    widechar;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    length = mblen(mb_string, MB_CUR_MAX);
    length = mbtowc(&widechar, mb_string + length, MB_CUR_MAX);
    printf("The wide character %lc has length of %d.\n", widechar, length);
  return  0;

  /*************************************************************************
       The output should be similar to :

       The wide character  B has length of 2.
  *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbstowcs — Convert Multibyte String to Wide-Character String" on page 391
"setlocale — Set Locale" on page 499
"wcslen — Calculate Length of Wide-Character String" on page 722
"wctomb — Convert Wide Character to Multibyte Character" on page 753
"<locale.h>" on page 766
"<stdlib.h>" on page 775

memccpy — Copy Bytes

Format

```
#include <string.h>        /* also in <memory.h> */
void *memccpy(void *dest, void *src, int c, unsigned cnt);
```

Description

Language Level: Extension

memccpy copies bytes from *src* to *dest*, up to and including the first occurrence of the character *c* or until *cnt* bytes have been copied, whichever comes first.

Return Value

If the character *c* is copied, memccpy returns a pointer to the byte in *dest* that immediately follows the character. If *c* is not copied, memccpy returns NULL.

This example copies up to 55 characters, or until it copies the '.' character, from the source to the buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[60];
char result[60];

int main(void)
{
    memcpy(source, "This is the string. This part won't be copied.", 55);
    if (NULL == memccpy(result, source, '.', 55)) {
        printf("Error in copying source.\n");
        exit(EXIT_FAILURE);
    }
    else
        printf("%s\n",  result);
    return  0;

    /*****************************************************************************
        The output should be:

        This is the string.
    *****************************************************************************/
}
```

"memchr — Search Buffer" on page 396
"memcmp — Compare Buffers" on page 397
"memcpy — Copy Bytes" on page 399
"memmove — Copy Bytes" on page 402
"memset — Set Bytes to Value" on page 403
"<memory.h>" on page 771
"<string.h>" on page 777

memchr — Search Buffer

Format

```
#include <string.h>    /* also in <memory.h> */
void *memchr(const void *buf, int c, size_t count);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

memchr searches the first *count* bytes of *buf* for the first occurrence of *c* converted to an unsigned character. The search continues until it finds *c* or examines *count* bytes.

Return Value

memchr returns a pointer to the location of *c* in *buf*. It returns NULL if *c* is not within the first *count* bytes of *buf*.

This example finds the first occurrence of "x" in the string that you provide. If it is found, the string that starts with that character is printed.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   char  *result;

    if (argc != 2)
        printf("Usage: %s string\n", argv[0]);
   else  {
        if ((result = (char *)memchr(argv[1], 'x', strlen(argv[1]))) != NULL)
            printf("The string starting with x is %s\n", result);
       else
            printf("The letter x cannot be found in the string\n");
   }
   return  0;

   /*************************************************************************
        If the program is passed the argument 'boxing', the output should be:

        The string starting with x is xing
   *************************************************************************/
}
```

"memccpy — Copy Bytes" on page 395
"memcmp — Compare Buffers" on page 397
"memcpy — Copy Bytes" on page 399
"memicmp — Compare Bytes" on page 400
"memmove — Copy Bytes" on page 402
"memset — Set Bytes to Value" on page 403
"strchr — Search for Character" on page 537
"<memory.h>" on page 771
"<string.h>" on page 777

memcmp — Compare Buffers

Format

```
#include <string.h>    /* also in <memory.h> */
int memcmp(const void *buf1, const void *buf2, size_t count);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

memcmp compares the first *count* bytes of *buf1* and *buf2*.

Return Value

memcmp returns a value indicating the relationship between the two buffers as follows:

Value	Meaning
Less than 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
Greater than 0	*buf1* greater than *buf2*

This example reports the relation between the two arguments passed to main to determine which, if either, is greater.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
    int  len;
    int  result;

     if (argc != 3) {
          printf("Usage: %s string1 string2\n", argv[0]);
    }
    else  {
```

memcmp

```
        /* Determine the length to be used for comparison                    */

        if (strlen(argv[1]) < strlen(argv[2]))
            len = strlen(argv[1]);
        else
            len = strlen(argv[2]);
        result = memcmp(argv[1], argv[2], len);
        printf("When the first %i characters are compared,\n", len);
        if (0 == result)
            printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
        else
            if (result < 0)

                printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
            else
                printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
    }
    return  0;

    /*****************************************************************************
        If the program is passed the arguments "firststring secondstring",
        the output should be:

        When the first 11 characters are compared,
        "firststring" is less than "secondstring"
    *****************************************************************************/
}
```

"memchr — Search Buffer" on page 396
"memcpy — Copy Bytes" on page 399
"memicmp — Compare Bytes" on page 400
"memmove — Copy Bytes" on page 402
"memset — Set Bytes to Value" on page 403
"strcmp — Compare Strings" on page 538
"<string.h>" on page 777

memcpy — Copy Bytes

Format

```
#include <string.h>    /* also <memory.h> */
void *memcpy(void *dest, const void *src, size_t count);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

memcpy copies *count* bytes of *src* to *dest*. The behavior is undefined if copying takes place between objects that overlap. (The memmove function allows copying between objects that may overlap.)

Return Value

memcpy returns a pointer to *dest*.

This example copies the contents of source to target.

```
#include <string.h>
#include <stdio.h>

#define   MAX_LEN           80
char source[MAX_LEN] = "This is the source string";
char target[MAX_LEN] = "This is the target string";

int main(void)
{
    printf("Before memcpy, target is \"%s\"\n", target);
    memcpy(target, source, sizeof(source));
    printf("After memcpy, target becomes \"%s\"\n", target);
   return  0;

  /*****************************************************************************
       The output should be:

       Before memcpy, target is "This is the target string"
       After memcpy, target becomes "This is the source string"
  *****************************************************************************/
}
```

"memccpy — Copy Bytes" on page 395
"memchr — Search Buffer" on page 396
"memcmp — Compare Buffers" on page 397
"memmove — Copy Bytes" on page 402
"memset — Set Bytes to Value" on page 403
"strcpy — Copy Strings" on page 544
"<string.h>" on page 777

memicmp — Compare Bytes

Format

```
#include <string.h>       /* also in <memory.h> */
int memicmp(void *buf1, void *buf2, unsigned int cnt);
```

Description

Language Level: Extension

memicmp compares the first *cnt* bytes of *buf1* and *buf2* without regard to the case of letters in the two buffers. The function converts all uppercase characters into lowercase and then performs the comparison.

Return Value

The return value of memicmp indicates the result as follows:

Value	Meaning
Less than 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
Greater than 0	*buf1* greater than *buf2*.

This example copies two strings that each contain a substring of 29 characters that are the same except for case. The example then compares the first 29 bytes without regard to case.

```
#include <stdio.h>
#include <string.h>

char first[100],second[100];

int main(void)
{
   int  result;

    strcpy(first, "Those Who Will Not Learn From History");
    strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
    printf("Comparing the first 29 characters of two strings.\n");
    result = memicmp(first, second, 29);
    printf("The first 29 characters of String 1 are ");
    if (result < 0)
        printf("less than String 2.\n");
   else
        if (0 == result)
            printf("equal to String 2.\n");
        else
            printf("greater than String 2.\n");
   return  0;

   /*****************************************************************************
        The output should be:

        Comparing the first 29 characters of two strings.
        The first 29 characters of String 1 are equal to String 2
   *****************************************************************************/
}
```

"memchr — Search Buffer" on page 396
"memcmp — Compare Buffers" on page 397
"memcpy — Copy Bytes" on page 399
"memmove — Copy Bytes" on page 402
"memset — Set Bytes to Value" on page 403
"strcmp — Compare Strings" on page 538
"<memory.h>" on page 771
"<string.h>" on page 777

memmove — Copy Bytes

Format

```
#include <string.h>    /* also in <memory.h> */
void *memmove(void *dest, const void *src, size_t count);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

memmove copies *count* bytes of *src* to *dest*. memmove allows copying between objects that may overlap. The behavior is as if the *src* is first copied into a temporary array.

Return Value

memmove returns a pointer to *dest*.

This example copies the word shiny from position target + 2 to position target + 8.

```
#include <string.h>
#include <stdio.h>

#define   SIZE                21
char target[SIZE] = "a shiny white sphere";

int main(void)
{
    char *p = target+8;                          /* p points at the starting character
                                                        of the word we want to replace     */
    char *source = target+2;                                 /* start of "shiny" */

    printf("Before memmove, target is \"%s\"\n", target);
    memmove(p, source, 5);
    printf("After memmove, target becomes \"%s\"\n", target);
   return  0;

   /*****************************************************************************
       The output should be:

       Before memmove, target is "a shiny white sphere"
       After memmove, target becomes "a shiny shiny sphere"
   *****************************************************************************/
}
```

“memccpy — Copy Bytes” on page 395
“memchr — Search Buffer” on page 396
“memcmp — Compare Buffers” on page 397
“memcpy — Copy Bytes” on page 399
“memset — Set Bytes to Value” on page 403
“strcpy — Copy Strings” on page 544
“<string.h>” on page 777

memset — Set Bytes to Value

Format

```
#include <string.h>    /* also in <memory.h> */
void *memset(void *dest, int c, size_t count);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

memset sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned character.

Return Value

memset returns a pointer to *dest*.

This example sets 10 bytes of the buffer to A and the next 10 bytes to B.

```
#include <string.h>
#include <stdio.h>

#define   BUF_SIZE          20

int main(void)
{
    char  buffer[BUF_SIZE+1];
    char  *string;

     memset(buffer, 0, sizeof(buffer));
     string = (char *)memset(buffer, 'A', 10);
     printf("\nBuffer contents: %s\n", string);
     memset(buffer+10, 'B', 10);
     printf("\nBuffer contents: %s\n", buffer);
    return  0;

   /*************************************************************************
        The output should be:

        Buffer contents: AAAAAAAAAA
        Buffer contents: AAAAAAAAAABBBBBBBBBB
   *************************************************************************/
}
```

"memccpy — Copy Bytes" on page 395
"memchr — Search Buffer" on page 396
"memcmp — Compare Buffers" on page 397
"memcpy — Copy Bytes" on page 399
"memicmp — Compare Bytes" on page 400
"memmove — Copy Bytes" on page 402
"strnset – strset — Set Characters in String" on page 574
"<memory.h>" on page 771
"<string.h>" on page 777

_mheap — Query Memory Heap for Allocated Object

Format

```
#include <umalloc.h>
Heap_t _mheap(void *ptr);
```

Description

Language Level: Extension

_mheap determines from which heap the object specified by *ptr* was allocated. The *ptr* must be a valid pointer that was returned from a runtime allocation function (_ucalloc, malloc, realloc, and so on). If the pointer is not valid, the results of _mheap are undefined.

For more information about creating and using heaps, see the chapter on Managing Memory in the *Programming Guide*.

Return Value

_mheap returns the handle of the heap from which the object was allocated. If the object was allocated from the runtime heap, _mheap returns _RUNTIME_HEAP. If the object passed to _mheap is NULL, _mheap returns NULL. If the object is not valid, _mheap either returns NULL or the results are unpredictable and an exception may occur.

This example allocates a block of memory from the heap, then uses _mheap to determine which heap the block came from.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   char    *ptr;

    if (NULL == (ptr = (char*)malloc(10))) {
        puts("Could not allocate memory block.");
       exit(EXIT_FAILURE);
   }
    printf("Handle of heap used is 0x%x\n", _mheap(ptr));
   return  0;

   /*************************************************************************
        The output should be similar to :

        Handle of heap used is 0x70000
   *************************************************************************/
}
```

"Managing Memory" in the *Programming Guide*
"_msize — Return Number of Bytes Allocated" on page 413
"_ucreate — Create a Memory Heap" on page 646
"_ustats — Get Information about Heap" on page 691

_mheap

"<umalloc.h>" on page 779

min — Return Lesser of Two Values

Format

```
#include <stdlib.h>
type min(type a, type b);
```

Description

Language Level: Extension

min compares two values and determines the smaller of the two. The data *type* can be any arithmetic data type, signed or unsigned. The *type* must be the same for both arguments to min.

Note: Because min is a macro, if the evaluation of the arguments contains side effects (post-increment operators, for example), the results of both the side effects and the macro will be undefined.

Return Value

min returns the smaller of the two values.

This example prints the smaller of the two values, a and b.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 21;

    printf("The smaller of %d and %d is %d\n", a, b, min(a, b));
   return  0;

   /*****************************************************************************
       The output should be:

       The smaller of 10 and 21 is 10
   *****************************************************************************/
}
```

"max — Return Larger of Two Values" on page 381
"<stdlib.h>" on page 775

mkdir — Create New Directory

Format

```
#include <direct.h>
int mkdir(char *pathname);
```

Description

Language Level: XPG4, Extension

mkdir creates a new directory with the specified *pathname*. Because only one directory can be created at a time, only the last component of *pathname* can name a new directory.

Note: In earlier releases of VisualAge C++, mkdir began with an underscore (_mkdir). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _mkdir to mkdir for you.

Return Value

mkdir returns the value 0 if the directory was created. A return value of -1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCESS	The directory was not created; the given name is the name of an existing file, directory, or device.
ENOENT	The *pathname* was not found.

This example creates two new directories: one at the root on drive C:, and one in the tmp subdirectory of the current working directory.

mkdir

```
#include <stdio.h>
#include <direct.h>
#include <string.h>

int main(void)
{
   char  *dir1,*dir2;

/*    Create the directory "aleng" in the root directory of the C: drive.          */

    dir1 = "c:\\aleng";
    if (0 == (mkdir(dir1)))
        printf("%s directory was created.\n", dir1);
   else
        printf("%s directory was not created.\n", dir1);

/*    Create the subdirectory "simon" in the current directory.                    */

    dir2 = "simon";
    if (0 == (mkdir(dir2)))
        printf("%s directory was created.\n", dir2);
   else
        printf("%s directory was not created.\n", dir2);

/*    Remove the directory "aleng" from the root directory of the C: drive.       */

    printf("Removing directory 'aleng' from the root directory.\n");
   if  (rmdir(dir1))
       perror(NULL);
   else
        printf("%s directory was removed.\n", dir1);

/*    Remove the subdirectory "simon" from the current directory.                  */

    printf("Removing subdirectory 'simon' from the current directory.\n");
   if  (rmdir(dir2))
       perror(NULL);
   else
        printf("%s directory was removed.\n", dir2);
   return  0;

   /*****************************************************************************
        The output should be:

        c:\aleng directory was created.
        simon directory was created.
        Removing directory 'aleng' from the root directory.
        c:\aleng directory was removed.
        Removing subdirectory 'simon' from the current directory.
        simon directory was removed.
   *****************************************************************************/
}
```


"chdir — Change Current Working Directory" on page 80
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276

“rmdir — Remove Directory” on page 467
“<direct.h>” on page 762

mktime — Convert Local Time

Format

```
#include <time.h>
time_t mktime(struct tm *time);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

mktime converts local time, stored as a tm structure pointed to by *time*, into a time_t structure suitable for use with other time functions. The values of some structure elements pointed to by *time* are not restricted to the ranges shown for "gmtime — Convert Time" on page 289.

The values of tm_wday and tm_yday passed to mktime are ignored and are assigned their correct values on return.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Return Value

mktime returns the calendar time having type time_t. The value (time_t)(-1) is returned if the calendar time cannot be represented.

This example prints the day of the week that is 40 days and 16 hours from the current date.

```
#include <stdio.h>
#include <time.h>

char *wday[] =    { "Sunday", "Monday", "Tuesday", "Wednesday",
                        "Thursday", "Friday", "Saturday" } ;

int main(void)
{
    time_t  t1,t3;
     struct tm *t2;

     t1 = time(NULL);
     t2 = localtime(&t1);
     t2->tm_mday += 40;
     t2->tm_hour += 16;
     t3 = mktime(t2);
     printf("40 days and 16 hours from now, it will be a %9s \n", wday[t2->tm_wday
         ]);
    return  0;

    /**********************************************************************
         The output should be similar to:

         40 days and 16 hours from now, it will be a Sunday
    **********************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47

"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"time — Determine Current Time" on page 625
"<time.h>" on page 779

modf — Separate Floating-Point Value

Format

```
#include <math.h>
double modf(double x, double *intptr);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

modf breaks down the floating-point value *x* into fractional and integral parts. The signed fractional portion of *x* is returned. The integer portion is stored as a double value pointed to by *intptr*. Both the fractional and integral parts are given the same sign as *x*.

Return Value

modf returns the signed fractional portion of *x*.

This example breaks the floating-point number –14.876 into its fractional and integral components.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y,d;

    x = -14.876;
    y = modf(x, &d);
    printf("x = %lf\n", x);
    printf("Integral part = %lf\n", d);
    printf("Fractional part = %lf\n", y);
   return  0;

   /*****************************************************************************
        The output should be:

        x = -14.876000
        Integral part = -14.000000
        Fractional part = -0.876000
   *****************************************************************************/
}
```

"fmod — Calculate Floating-Point Remainder" on page 217
"frexp — Separate Floating-Point Value" on page 244
"ldexp — Multiply by a Power of Two" on page 340
"<math.h>" on page 770

_msize — Return Number of Bytes Allocated

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
size_t _msize(void *ptr)
```

Description

Language Level: Extension

_msize determines the number of bytes that were allocated to the pointer argument *ptr*. The *ptr* must have been returned from one of the runtime memory allocation functions (_ucalloc, malloc, and so on).

You cannot pass the argument of an object that has been freed.

Return Value

_msize returns the number of bytes allocated. If the argument is not a valid pointer returned from a memory allocation function, the return value is undefined. If NULL is passed, _msize returns 0.

This example displays the size of an allocated object from malloc.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char  *ptr;

    if (NULL == (ptr = (char*)malloc(10))) {
        puts("Could not allocate memory block.");
       exit(EXIT_FAILURE);
   }
    memset(ptr, 'x', 5);
    printf("The size of the allocated object is %u.\n",_msize(ptr));
   return  0;

   /*****************************************************************************
        The output should be similar to :

        The size of the allocated object is 10.
   *****************************************************************************/
}
```

"calloc — Reserve and Initialize Storage" on page 75
"malloc — Reserve Storage Block" on page 376
"realloc — Change Reserved Storage Block Size" on page 452
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

nl_langinfo — Retrieve Locale Information

Format

```
#include <nl_types.h>
#include <langinfo.h>
char *nl_langinfo(nl_item item);
```

Description

Language Level: XPG4

nl_langinfo retrieves from the current locale the string that describes the requested information specified by *item*.

The constant names and values for *item* are defined in <langinfo.h>. For a list of macros that define the constants used to identify the information queried in the current locale, see the table of defined macros under "<langinfo.h>" on page 764.

You cannot retrieve the following information for the current locale:

```
t_fmt_ampm
era
era_year
era_d_fmt
alt_digits
t_fmt_ampm
alt_digits
```

Return Value

nl_langinfo returns a pointer to a null-terminated string containing information about the active language or cultural area. The active language or cultural area is determined by the most recent setlocale call. Subsequent calls to the *setlocale* function may modify the string that the return value points to. Your own code cannot modify the array.

If *item* is not valid, nl_langinfo returns a pointer to an empty string.

This example uses nl_langinfo to retrieve the current codeset name.

```
#include <nl_types.h>
#include <langinfo.h>
#include <stdio.h>

int main(void)
{
    printf("Current codeset is %s\n", nl_langinfo(CODESET));
   return  0;

   /*************************************************************************
        The output should be similar to :

        Current codeset is IBM-850
   *************************************************************************/
}
```

"localdtconv — Return Date and Time Formatting Convention" on page 350
"localeconv — Retrieve Information from the Environment" on page 352
"setlocale — Set Locale" on page 499
"<langinfo.h>" on page 764
"<nl_types.h>" on page 771

_onexit — Record Termination Function

Format

```
#include <stdlib.h>
onexit_t _onexit(onexit_t func);
```

Description

Language Level: Extension

_onexit records the address of a function *func* to call when the program ends normally. Successive calls to _onexit create a stack of functions that run in a last-in, first-out order. The functions passed to _onexit cannot take parameters.

You can record up to 32 termination functions with calls to _onexit and atexit. If you exceed 32 functions, _onexit returns the value NULL.

Note: For portability, use the ANSI/ISO standard atexit function, which is equivalent to _onexit.

Return Value

If successful, _onexit returns a pointer to the function; otherwise, it returns a NULL value.

This example specifies and defines four distinct functions that run consecutively at the completion of main.

```
#include <stdio.h>
#include <stdlib.h>

int fn1(void)
{
   printf("next.\n");
   return  0;
}

int fn2(void)
{
   printf("run  ");
   return  0;
}

int fn3(void)
{
   printf("is  ");
   return  0;
}
```

```
int fn4(void)
{
    printf("This  ");
    return  0;
}

int main(void)
{
    _onexit((onexit_t)fn1);
    _onexit((onexit_t)fn2);
    _onexit((onexit_t)fn3);
    _onexit((onexit_t)fn4);
     printf("This is run first.\n");
    return  0;

    /*************************************************************************
        The output should be:

        This is run first.
        This is run next.
    *************************************************************************/
}
```

"abort — Stop a Program" on page 39
"atexit — Record Program Termination Function" on page 54
"exit — End Program" on page 179
"_exit — End Process" on page 180
"<stdlib.h>" on page 775

open — Open File

Format

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
int open(char *pathname, int oflag, int pmode);
```

Description

Language Level: XPG4, Extension

open opens the file specified by *pathname* and prepares the file for subsequent reading or writing as defined by *oflag*. open can also prepare the file for reading and writing.

The *oflag* is an integer expression formed by combining one or more of the following constants, defined in <fcntl.h>. To specify more than one constant, join the constants with the bitwise OR operator (|); for example, O_CREAT | O_TEXT.

Oflag	Meaning
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file. This flag has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This flag applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.
O_RDWR	Open the file for reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TRUNC	Open and truncate an existing file to 0 length. The file must have write permission. The contents of the file are destroyed, and O_TRUNC cannot be specified with O_RDONLY.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.
O_BINARY	Open the file in binary (untranslated) mode.
O_TEXT	Open the file in text (translated) mode.

If neither O_BINARY or O_TEXT is specified, the default will be O_TEXT; it is an error to specify both O_BINARY and O_TEXT. You must specify one of the access mode flags, O_RDONLY, O_WRONLY, or O_RDWR. There is no default.
Warning: Use O_TRUNC with care; it destroys the complete contents of an existing file.

For more details on text and binary modes and their differences, see "Stream Processing" in the *Programming Guide*.

The *pmode* argument is an integer expression containing one or both of the constants S_IWRITE and S_IREAD, defined in <sys\stat.h>. The *pmode* is required only when O_CREAT is specified. If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the permission settings for the file. These are set when the new file is closed for the first time. The meaning of the *pmode* argument is as follows:

Value	Meaning
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If write permission is not given, the file is read-only. Under the Windows operating system, all files are readable; you cannot give write-only permission. The modes S_IWRITE and S_IREAD | S_IWRITE are equivalent.

open applies the current file permission mask to *pmode* before setting the permissions. (See "umask — Sets File Mask of Current Process" on page 679.)

Note: In earlier releases of VisualAge C++, open began with an underscore (_open). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _open to open for you.

Return Value

open returns a file handle for the opened file. A return value of –1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EACCESS	The given *pathname* is a directory; or the file is read-only but an open for writing was attempted; or a sharing violation occurred.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available.
EINVAL	An incorrect argument was passed.
ENOENT	The file or *pathname* were not found.
EOS2ERR	The call to the operating system was not successful.

This example opens the file edopen.dat by creating it as a new file, truncating it if it exists, and opening it so it can be read and written to. The open command issued also grants permission to read from and write to the file.

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

int main(void)
{
    int  fh;

     if (-1 == (fh = open("edopen.dat", O_CREAT|O_TRUNC|O_RDWR,
                                    S_IREAD|S_IWRITE)))  {
          perror("Unable to open edopen.dat");
          return  EXIT_FAILURE;
    }
     printf("File was successfully opened.\n");
     if (-1 == close(fh)) {
          perror("close  error");
          return  EXIT_FAILURE;
    }
    return  0;

   /*****************************************************************************
         The output should be:

         File was successfully opened.
   *****************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"creat — Create New File" on page 100
"fdopen — Associates Input Or Output With File" on page 194
"fopen — Open Files" on page 218
"_sopen — Open Shared File" on page 516
"umask — Sets File Mask of Current Process" on page 679
"<fcntl.h>" on page 763
"<io.h>" on page 764
"<sys\stat.h>" on page 778

_outp — Write Byte to Output Port

Format

```
#include <conio.h>    /* also in <builtin.h> */
int _outp( const unsigned int port, const int value );
```

Description

Language Level: Extension

_outp writes a byte value to the specified *port*. The *port* number must be an unsigned integer value within the range 0 to 65 535 inclusive. The byte *value* must be within the range 0 to 255 inclusive.

Note: _outp is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _outp.
You cannot parenthesize a call to _outp. (Parentheses specify a call to the function's backing code, and _outp has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_outp returns the integer value that was output to the specified *port*. There is no error return value, and _outp does not set errno.

This example uses _outp to write a byte to a specified output port and return the data written.

_outp

```
#include <builtin.h>

#define LOWER    0
#define UPPER1 255
#define UPPER2 65535

int Add1(int j);

static int g;
enum fruit {apples=10, bananas, cantaloupes};
enum fruit f = cantaloupes;
int arr[] = {cantaloupes, bananas, apples};
struct
{
    int    i;
    char  ch;
} st;

int main(void)
{
     static int i;
     volatile const int c = 0;

     st.i = c - bananas;
     g = _outp(LOWER,apples);
     i = _outp(255, 0);
                                                /* ============================== */
                                                /* Types of port number passed : */
                                                /* ------------------------------ */
     i = _outp(UPPER2,UPPER1);                 /* - #define constant                */
     i = _outp(st.i, bananas);                 /* - element of structure           */
     i = _outp(f,arr[1]);                      /* - enumerated variable            */
     i = _outp(_inp(arr[2]),apples);           /* - return value from a            */
                                                /*      builtin function call       */
                                                /* ------------------------------ */
     i = _outp(_outp(apples,Add1(LOWER)),6);
    return  0;
}

int Add1(int j)
{
     j += 1;
    return  j;
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpw — Read Unsigned Short from Input Port" on page 312
"_inpd — Read Doubleword from Input Port" on page 310
"_outpw — Write Word to Output Port" on page 425
"_outpd — Write Double Word to Output Port" on page 423
"<builtin.h>" on page 761
"<conio.h>" on page 762

_outpd — Write Double Word to Output Port

Format

```
#include <conio.h>    /* also in <builtin.h */
unsigned long _outpd( const unsigned int port, const unsigned long value );
```

Description

Language Level: Extension

_outpd writes an unsigned long *value* to the specified *port*. The *port* number must be a value within the range 0 to 65 535 inclusive. The unsigned long *value* must be within the range 0 to 4 294 967 295 inclusive.

Note: _outpd is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _outpd.
You cannot parenthesize a call to _outpd. (Parentheses specify a call to the function's backing code, and _outpd has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_outpd returns the unsigned long value that was output to the specified *port*. There is no error return value, and _outpd does not set errno.

This example uses _outpd to write a doubleword value to a specified output port and return the data written.

_outpd

```
#include <builtin.h>

#define LOWER    0
#define UPPER1 65535
#define UPPER2 4294967295

int Add1(int j);

volatile long g;
enum fruit {apples=10, bananas, cantaloupes};
enum fruit f = cantaloupes;
int arr[] = {cantaloupes, bananas, apples};
union
{
    volatile  int    i;
     volatile char ch;
} un;

int main(void)
{
     unsigned long l;
     volatile const short c = 0;

     un.i = bananas * f;
     g = _outpd(0,LOWER);
                                                /* ============================== */
                                                /* Types of port number passed : */
                                                /* ------------------------------ */
     l = _outpd(UPPER1, UPPER2);              /* - #define constant                  */
     l = _outpd(un.i ,f);                     /* - element of union                  */
     l = _outpd(Add1(c), apples);             /* - return value from a               */
                                                /*      function  call                 */
                                                /* ------------------------------ */
     l = _outpd(_outpw(255,Add1(LOWER)),6);
    return  0;
}

int Add1(int j)
{
     j += 1;
    return  j;
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpw — Read Unsigned Short from Input Port" on page 312
"_inpd — Read Doubleword from Input Port" on page 310
"_outp — Write Byte to Output Port" on page 421
"_outpw — Write Word to Output Port" on page 425
"<builtin.h>" on page 761
"<conio.h>" on page 762

_outpw — Write Word to Output Port

Format

#include <conio.h> /* also in <builtin.h */
unsigned short _outpw(const unsigned int *port*, const unsigned short *word*);

Description

Language Level: Extension

_outpw writes an unsigned short *word* to the specified *port*. The *port* number must be an unsigned integer value within the range 0 to 65 535 inclusive. The unsigned short *word* must be in the range 0 to 65 535.

Note: _outpw is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of _outpw.
You cannot parenthesize a call to _outpw. (Parentheses specify a call to the function's backing code, and _outpw has none.)

You can run code containing this function only at ring zero. Otherwise, an invalid instruction exception is generated.

Return Value

_outpw returns the value that was output to the specified *port*. There is no error return value, and _outpw does not set errno.

This example uses _outpw to write an unsigned short value to a specified output port and return the data written.

_outpw

```
#include <builtin.h>

#define LOWER 0
#define UPPER 65535

int Add1(int j);

unsigned int g;

int main(void)
{
    enum fruit {apples = 10, bananas, cantaloupes};
    enum fruit f = cantaloupes;
    int arr[] = {cantaloupes, bananas, apples};
    unsigned short s;
    static int i = 0;
    volatile const int c = 255;

    g = _outpw(cantaloupes, i);
                                        /* ================================ */
                                        /* Types of port number passed : */
                                        /* ------------------------------ */
    s = _outpw(UPPER, LOWER);           /* - #define constant                */
    s = _outpw(c, Add1(255));           /* - constant                        */
    s = _outpw(_inpw(arr[2]),apples); /* - return value from a               */
                                        /*      builtin function call        */
                                        /* ------------------------------ */
    s = _outpw(_outpw(bananas ,UPPER),6);
   return  0;
}

int Add1(int j)
{
    j += 1;
   return  j;
}
```

"_inp — Read Byte from Input Port" on page 308
"_inpw — Read Unsigned Short from Input Port" on page 312
"_inpd — Read Doubleword from Input Port" on page 310
"_outp — Write Byte to Output Port" on page 421
"_outpd — Write Double Word to Output Port" on page 423
"<builtin.h>" on page 761
"<conio.h>" on page 762

perror — Print Error Message

Format

```
#include <stdio.h>
void perror(const char *string);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

perror prints an error message to **stderr**. If *string* is not NULL and does not point to a null character, the string pointed to by *string* is printed to the standard error stream, followed by a colon and a space. The message associated with the value in errno is then printed followed by a new-line character.

To produce accurate results, you should ensure that perror is called immediately after a library function returns with an error; otherwise, subsequent calls may alter the errno value.

Return Value

There is no return value.

This example tries to open a stream. If fopen fails, the example prints a message and ends the program.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   FILE  *fh;

    if (NULL == (fh = fopen("myfile.mjq", "r"))) {
         perror("Could not open data file");
        abort();
   }
   return  0;

   /*****************************************************************************
        The output should be:

        Could not open data file: The file cannot be found.
   *****************************************************************************/
}
```

"clearerr — Reset Error Indicators" on page 87
"ferror — Test for Read/Write Errors" on page 198
"strerror — Set Pointer to Runtime Error Message" on page 550
"_strerror — Set Pointer to System Error String" on page 551
"<stdio.h>" on page 774

pow — Compute Power

Format

```
#include <math.h>
double pow(double x, double y);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

pow calculates the value of *x* to the power of *y*.

Return Value

If *y* is 0, pow returns the value 1. If *x* is 0 and *y* is negative, pow sets errno to EDOM and returns 0. If both *x* and *y* are 0, or if *x* is negative and *y* is not an integer, pow sets errno to EDOM, and returns 0.

If an overflow results, pow sets errno to ERANGE and returns +HUGE_VAL for a large result or -HUGE_VAL for a small result.

This example calculates the value of 2^3.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  x,y,z;

    x = 2.0;
    y = 3.0;
    z = pow(x, y);
    printf("%lf to the power of %lf is %lf\n", x, y, z);
   return  0;

   /*************************************************************************
        The output should be:

        2.000000 to the power of 3.000000 is 8.000000
   *************************************************************************/
}
```

"exp — Calculate Exponential Function" on page 181
"_fsqrt — Calculate Square Root" on page 254
"log — Calculate Natural Logarithm" on page 358
"log10 — Calculate Base 10 Logarithm" on page 359
"sqrt — Calculate Square Root" on page 527
"<math.h>" on page 770

printf — Print Formatted Characters

Format

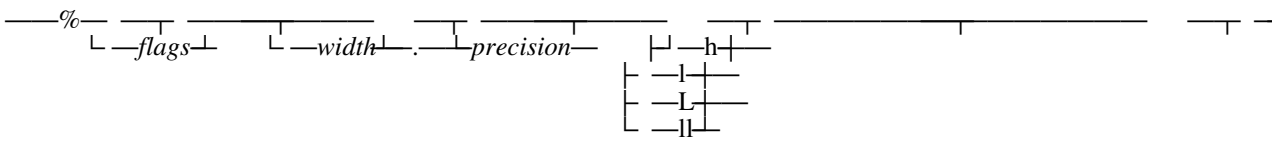
```
#include <stdio.h>
int printf(const char *format-string, argument-list);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

printf formats and prints a series of characters and values to the standard output stream **stdout**. The *format-string* consists of ordinary characters, escape sequences, and format specifications. The ordinary characters are copied in order of their appearance to **stdout**. Format specifications, beginning with a percent sign (%), determine the output format for any *argument-list* following the *format-string*.

The *format-string* is read left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. A format specification has the following form:



Each field of the format specification is a single, or double character, or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

The following optional fields control other aspects of the formatting:

Field	Description
flags	Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for wchar_t precision unit.
width	Minimum number of characters (bytes) output.
precision	Maximum number of characters (bytes) printed for all or part of the output field, or minimum number of digits printed for integer values.
h, l, L, ll	Size of argument expected:

- **h** A prefix with the integer types d, i, o, u, x, X, and n that specifies that the argument is short int or unsigned short int.
- **L** A prefix with e, E, f, g, or G types that specifies that the argument is long double.
- **l** A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long int or unsigned long int.
- **ll** A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long long int or unsigned long long int.

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to **stdout**. For example, to print a percent sign character, use %%.

In extended mode, printf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See “Infinity and NaN Support” on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, printf prints (null). (In previous releases of VisualAge C++, printf produced no output for a null string.)

The *type* characters and their meanings are given in the following table:

Character	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.

Character	Argument	Output Format
X	Integer	Unsigned hexadecimal integer, using ABCDEF.
f	Double	Signed value having the form [-]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point is equal to the requested precision. NaN and infinity values are printed in lowercase (nan and infinity).
F	Double	In extended mode (/Se option), identical to the f format except that NaN and infinity values are printed in uppercase (NAN and INFINITY). In modes other than extended, F is treated like any other character not included in this table.
e	Double	Signed value having the form [-]*d.dddd*e[*sign*]*ddd*, where *d* is a single-decimal digit, *dddd* is one or more decimal digits, *ddd* is 2 or 3 decimal digits, and *sign* is + or −.
E	Double	Identical to the e format except that E introduces the exponent instead of e.
g	Double	Signed value printed in f or e format. The e format is used only when the exponent of the value is less than -4 or greater than *precision*. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the g format except that E introduces the exponent (where appropriate) instead of e.
c	Character	Single character.
lc	Wide character	Multibyte character (converted as if by a call to wcrtomb).
s	String	Characters printed up to the first null character (\0) or until *precision* is reached. If you specify a null string, (null) is printed.
ls	Wide-character string.	Multibyte characters, printed up to the first wchar_t null character (L\0) is encountered in the wide-character string, or until the specified precision is reached. Conversion takes place as if by a call to wcrtomb. The displayed result does not include the terminating null character. If you do not specify the precision, you must end the wide-character string with a null character. A partial multibyte character cannot be written. If you specify a null string, (null) is printed.
n	Pointer to integer	Number of characters successfully written so far to the *stream* or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters.

The *flag* characters and their meanings are as follows (notice that more than one *flag* can appear in a format specification):

Table 3. Flag Characters

Flag	Meaning	Default
–	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix.
	When used with the f, F, e, or E formats, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G formats, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it; trailing zeros are truncated.
	When used with the ls format, the # flag causes precision to be measured in wchar_t characters.	Precision indicates the maximum number of bytes to be output.
0	When used with the d, i, o, u, x, X, e, E, f, F g, or G formats, the 0 flag causes leading 0's to pad the output to the field width. The 0 flag is ignored if precision is specified for an integer or if the – flag is specified.	Space padding.

The # flag should not be used with c, lc, d, i, u, s, or p types.

Width is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

Width never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

For the ls type, *width* is specified in bytes. If the number of bytes in the output value is less than the specified width, single-byte blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

The *width* specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list.

Precision is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

The *precision* specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in the following table:

Type	Meaning	Default
i d u o x X	*Precision* specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	If *precision* is 0 or omitted entirely, or if the period (.) appears without a number following it, the *precision* is set to 1.
f F e E	*Precision* specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.	Default *precision* is six. If *precision* is 0 or the period appears without a number following it, no decimal point is printed.
g G	*Precision* specifies the maximum number of significant digits printed.	All significant digits are printed.
c	No effect.	The character is printed.
lc	No effect.	The wchar_t character is printed.

Type	Meaning	Default
s	*Precision* specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed.	Characters are printed until a null character is encountered.
ls	*Precision* specifies the maximum number of bytes to be printed. Bytes in excess of *precision* are not printed; however, multibyte integrity is always preserved.	wchar_t characters are printed until a null character is encountered.

Return Value The printf function returns the number of bytes printed.

This example prints data in a variety of formats.

```
#include <stdio.h>

int main(void)
{
    char ch = 'h',*string = "computer";
    int count = 234,hex = 0x10,oct = 010,dec = 10;
    double fp = 251.7366;

   printf("%d    %+d      %06d       %X      %x       %o\n\n",  count,  count,  count,  count
        , count, count);
   printf("1234567890123%n4567890123456789\n\n",  &count);
    printf("Value of count should be 13; count = %d\n\n", count);
    printf("%10c%5c\n\n", ch, ch);
    printf("%25s\n%25.4s\n\n", string, string);
    printf("%f      %.2f      %e      %E\n\n",  fp,  fp,  fp,  fp);
    printf("%i       %i        %i\n\n", hex, oct, dec);
   return  0;

   /*****************************************************************************
        The output should be:

       234     +234      000234       EA      ea       352

       12345678901234567890123456789

        Value of count should be 13; count = 13

                    h      h

                              computer
                                   comp

        251.736600      251.74      2.517366e+02      2.517366E+02

       16      8       10
   *****************************************************************************/
}
```

"_cprintf — Print Characters to Screen" on page 98
"fprintf — Write Formatted Data to a Stream" on page 224
"fscanf — Read Formatted Data" on page 245
"scanf — Read Data" on page 486
"sprintf — Print Formatted Data to Buffer" on page 525
"sscanf — Read Data" on page 530
"swprintf — Format and Write Wide Characters to Buffer" on page 609
"vfprintf — Print Argument Data to Stream" on page 698
"vprintf — Print Argument Data" on page 700
"vsprintf — Print Argument Data to Buffer" on page 702
"vswprintf — Format and Write Wide Characters to Buffer" on page 704
"<stdio.h>" on page 774

putc – putchar — Write a Character

Format

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

putc converts *c* to unsigned char and then writes *c* to the output *stream* at the current position. putchar is equivalent to putc(*c*, stdout).

putc is equivalent to fputc except that, if it is implemented as a macro, putc can evaluate *stream* more than once. Therefore, the *stream* argument to putc should not be an expression with side effects.

Return Value

putc and putchar return the character written. A return value of EOF indicates an error.

This example writes the contents of a buffer to a data stream. In this example, the body of the for statement is null because the example carries out the writing operation in the test expression.

```
#include <stdio.h>

#define   LENGTH 80

int main(void)
{
     FILE *stream = stdout;
     int i, ch;
     char buffer[LENGTH + 1] = "Hello world";
     /* This could be replaced by using the fwrite routine */
     for ( i = 0;
             (i < sizeof(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
              ++i);
    return  0;


/*************************************************************************
Output should be:

Hello world
*************************************************************************/
}
```

"fputc — Write Character" on page 227
"_fputchar — Write Character" on page 229
"fwrite — Write Items" on page 263
"getc – getchar — Read a Character" on page 270
"_putch — Write Character to Screen" on page 438
"puts — Write a String" on page 441

"write — Writes from Buffer to File" on page 759
"<stdio.h>" on page 774

_putch — Write Character to Screen

Format

```
#include <conio.h>
int _putch(int c);
```

Description

Language Level: Extension

_putch writes the character *c* directly to the screen.

Return Value

If successful, _putch returns *c*. If an error occurs, _putch returns EOF.

This example defines a function gchar that is similar to _getche using the _putch and _getch functions:

```
#include <conio.h>

int gchar(void)
{
   int  ch;

    ch = _getch();
   _putch(ch);
   return  (ch);
}
```

"_cputs — Write String to Screen" on page 99
"_cprintf — Print Characters to Screen" on page 98
"fputc — Write Character" on page 227
"_getch - _getche — Read Character from Keyboard" on page 272
"putc – putchar — Write a Character" on page 436
"puts — Write a String" on page 441
"write — Writes from Buffer to File" on page 759
"<conio.h>" on page 762

putenv — Modify Environment Variables

Format

```
#include <stdlib.h>
int putenv(char *envstring);
```

Description

Language Level: XPG4, Extension

putenv adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process runs (for example, the default search path for libraries to be linked with a program).

The *envstring* argument must be a pointer to a string with the form:

```
varname=string
```

where *varname* is the name of the environment variable to be added or modified and *string* is the value of the variable. See the **Notes** below.

If *varname* is already part of the environment, *string* replaces its current value; if not, the new *varname* is added to the environment with the value *string*. To set a variable to an empty value, specify an empty *string*. A variable can be removed from the environment by specifying *varname* only, for example:

```
putenv("PATH");
```

Do not free the *envstring* pointer while the entry it points to is in use, or the environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to putenv and then exit from the function in which the variable is declared. Once you have added the *envstring* with putenv, any change to the entry it points to changes the environment used by your program.

The environment manipulated by putenv is local to the process currently running, that is, changes are local to the run-time environment to which the call to putenv is made. For example, if you have an *exe* and a *dll* both linking statically to the run-time library, if you call putenv in the *exe*, the changes will not be reflected in the *dll*.

You cannot enter new items in your command-level environment using putenv. When the program ends, the environment reverts to the parent process environment. This environment is passed on to some child processes created by the _spawn, exec, or system functions, and they get any new environment variables added using putenv.

GetEnvironmentVariable will not reflect any changes made using putenv, but getenv will reflect the changes.

Notes:

1. putenv can change the value of **_environ**, thus invalidating the envp argument to the main function.
2. You cannot use *%envvar%*, where *envvar* is any Windows environment variable, with putenv to concatenate new *envstring* and old *envstring*.
3. In earlier releases of VisualAge C++, putenv began with an underscore (_putenv). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _putenv to putenv for you.

Return Value

putenv returns 0 if it is successful. A return value of –1 indicates an error.

This example tries to change the environment variable PATH, and then uses getenv to get the current path. If the call to putenv fails, the example writes an error message.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *pathvar;

    if (-1 == putenv("PATH=a:\\bin;b:\\andy")) {
        printf("putenv failed - out of memory\n");
       return  EXIT_FAILURE;
   }

    /* getting and printing the current environment path                    */

    pathvar = getenv("PATH");
    printf("The current path is: %s\n", pathvar);
   return  0;

   /*****************************************************************************
        The output should be:

        The current path is: a:\bin;b:\andy
   *****************************************************************************/
}
```

"execl - _execvpe — Load and Run Child Process" on page 175
"getenv — Search for Environment Variables" on page 279
"_spawnl - _spawnvpe — Start and Run Child Processes" on page 519
"system — Invoke the Command Processor" on page 613
"envp Parameter to main" in the *Programming Guide*
"<stdlib.h>" on page 775

puts — Write a String

Format

```
#include <stdio.h>
int puts(const char *string);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

puts writes the given *string* to the standard output stream **stdout**; it also appends a new-line character to the output. The terminating null character is not written.

Return Value

puts returns EOF if an error occurs. A nonnegative return value indicates that no error has occurred.

This example writes Hello World to **stdout**.

```
#include <stdio.h>

int main(void)
{
    if (EOF == puts("Hello World"))
        printf("Error in puts\n");
   return 0;

   /*****************************************************************************
        The output should be:

       Hello World
   *****************************************************************************/
}
```

"_cputs — Write String to Screen" on page 99
"fputs — Write String" on page 230
"gets — Read a Line" on page 281
"putc – putchar — Write a Character" on page 436
"<stdio.h>" on page 774

putwc — Write Wide Character

Format

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
```

Description

Language Level: ANSI 93, XPG4

putwc converts the wide character *wc* to a multibyte character, and writes it to the *stream* at the current position. It also advances the file position indicator for the stream appropriately.

putwc function is equivalent to fputwc except that, if it is implemented as a macro, putwc can evaluate *stream* more than once. Therefore, the *stream* argument to putwc should not be an expression with side effects.

The behavior of putwc is affected by the LC_CTYPE category of the current locale. Using a non-wide-character function with putwc on the same stream results in undefined behavior.

After calling putwc, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling putwc.

Return Value

putwc returns the wide character written. If a write error occurs, putwc sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, putwc sets **errno** to EILSEQ and returns WEOF.

The following example uses putwc to convert the wide characters in wcs to multibyte characters and write them to the file putwc.out.

```
#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
    FILE       *stream;
     wchar_t *wcs = L"A character string.";
    int         i;

     if (NULL == (stream = fopen("putwc.out", "w"))) {
          printf("Unable to open: \"putwc.out\".\n");
         exit(EXIT_FAILURE);
    }

     for (i = 0; wcs[i] != L'\0'; i++) {
          errno = 0;
          if (WEOF == putwc(wcs[i], stream)) {
               printf("Unable to putwc() the wide character.\n"
                         "wcs[%d] = 0x%lx\n", i, wcs[i]);
               if (EILSEQ == errno)
                    printf("An invalid wide character was encountered.\n");
              exit(EXIT_FAILURE);
          }
    }
    fclose(stream);
    return  0;

    /*****************************************************************************
          The output file putwc.out should contain :

          A character string.
    *****************************************************************************/
}
```

“fputc — Write Character” on page 227
“_fputchar — Write Character” on page 229
“fputwc — Write Wide Character” on page 232
“getwc — Read Wide Character from Stream” on page 285
“putc – putchar — Write a Character” on page 436
“_putch — Write Character to Screen” on page 438
“putwchar — Write Wide Character to stdout” on page 444
“<stdio.h>” on page 774
“<wchar.h>” on page 780

putwchar — Write Wide Character to stdout

Format

```
#include <wchar.h>
wint_t putwchar(wint_t wc);
```

Description

Language Level: ANSI 93, XPG4

putwchar converts the wide character *wc* to a multibyte character and writes it to **stdout**. A call to putwchar is equivalent to putwc(*wc*, stdout).

The behavior of putwchar is affected by the LC_CTYPE category of the current locale. Using a non-wide-character function with putwchar on the same stream results in undefined behavior.

After calling putwchar, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling putwchar.

Return Value

putwchar returns the wide character written. If a write error occurs, putwchar sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, putwchar sets **errno** to EILSEQ and returns WEOF.

This example uses putwchar to write the string in wcs.

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
    wchar_t *wcs = L"A character string.";
   int         i;

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwchar(wcs[i])) {
            printf("Unable to putwchar() the wide character.\n");
            printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(EXIT_FAILURE);
        }
   }
   return  0;

   /*****************************************************************************
        The output should be similar to :

        A character string.
   *****************************************************************************/
}
```

"fputc — Write Character" on page 227
"_fputchar — Write Character" on page 229
"fputwc — Write Wide Character" on page 232
"getwchar — Get Wide Character from stdin" on page 287
"putc – putchar — Write a Character" on page 436
"_putch — Write Character to Screen" on page 438
"<wchar.h>" on page 780

qsort — Sort Array

Format

```
#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *key, const void *element));
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

qsort sorts an array of *num* elements, each of *width* bytes in size. The *base* pointer is a pointer to the array to be sorted. qsort overwrites this array with the sorted elements.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. qsort calls this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values:

Value	Meaning
Less than 0	*key* less than *element*
0	*key* equal to *element*
Greater than 0	*key* greater than *element*

The sorted array elements are stored in ascending order, as defined by your *compare* function. You can sort in reverse order by reversing the sense of "greater than" and "less than" in *compare*. The order of the elements is unspecified when two elements compare equally.

Return Value There is no return value.

This example sorts the arguments (argv) in ascending lexical sequence, using the comparison function compare() supplied in the example.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* ------------------------------------------------------------- */
/* compare() routine called internally by qsort()                */
/*                                                               */
/* Assert: Library always calls functions internally with        */
/*          _Optlink linkage convention. Ensure that compare() is */
/*          always  _Optlink.                                    */
/* ------------------------------------------------------------- */

int _Optlink compare(const void *arg1,const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}

int main(int argc,char *argv[])
{
   int  i;
   argv++;
   argc--;

    qsort((char *)argv, argc, sizeof(char *), compare);
    for (i = 0; i < argc; ++i)
        printf("%s\n",  argv[i]);
   return  0;

   /*************************************************************************
        Assuming command line of: qsort kent theresa andrea laura brenden
        Output should be:

          andrea
          brenden
          kent
          laura
          theresa
   *************************************************************************/
}
```

"bsearch — Search Arrays" on page 71
"lfind - lsearch — Find Key in Array" on page 342
"<search.h>" on page 772
"<stdlib.h>" on page 775

raise — Send Signal

Format

```
#include <signal.h>
int raise(int sig);
```

Description

Language Level: ANSI, SAA, XPG4

raise sends the signal *sig* to the running program. You can then use signal to handle *sig*.

Signals and signal handling are described in "signal — Handle Interrupt Signals" on page 510, and in the *Programming Guide* under Signal and Exception Handling.

Return Value

raise returns 0 if successful, nonzero if unsuccessful.

This example establishes a signal handler called sig_hand for the signal SIGUSR1. The signal handler is called whenever the SIGUSR1 signal is raised and will ignore the first nine occurrences of the signal. On the tenth raised signal, it exits the program with an error code of 10. Note that the signal handler must be reestablished each time it is called.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sig_hand(int);             /* declaration of sig_hand() as a function              */
int main(void)
{
    signal(SIGUSR1, (_SigFunc)sig_hand);      /* set up handler for SIGUSR1          */

    raise(SIGUSR1);                           /* signal SIGUSR1 is raised             */
                                /* sig_hand() is called                               */
   return  0;
}

void sig_hand(int sig)
{
    static int count = 0;                     /* initialized only once                */
   count++;

    if (10 == count)        /* ignore the first 9 occurrences of this signal          */
        exit(10);
   else
         signal(SIGUSR1,(_SigFunc)sig_hand);                  /* set up the handler again  */
   raise(SIGUSR1);
}
```

"signal — Handle Interrupt Signals" on page 510
Signal and Exception Handling in the *Programming Guide*
"<signal.h>" on page 773

rand — Generate Random Number

Format

```
#include <stdlib.h>
int rand(void);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

rand generates a pseudo-random integer in the range 0 to RAND_MAX (macro defined in <stdlib.h>). Use srand before calling rand to set a starting point for the random number generator. If you do not call srand first, the default seed is 1.

Return Value

rand returns a pseudo-random number.

This example prints the first 10 pseudo-random numbers generated.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int  x;

    for (x = 1; x <= 10; x++)
        printf("iteration %d, rand=%d\n", x, rand());
   return  0;

   /*************************************************************************
        The output should be:

        iteration 1, rand=16838
        iteration 2, rand=5758
        iteration 3, rand=10113
        iteration 4, rand=17515
        iteration 5, rand=31051
        iteration 6, rand=5627
        iteration 7, rand=23010
        iteration 8, rand=7419
        iteration 9, rand=16212
        iteration 10, rand=4086
   *************************************************************************/
}
```

"srand — Set Seed for rand Function" on page 528
"<stdlib.h>" on page 775

read — Read Into Buffer

Format

#include <io.h>
int read(int *handle*, char **buffer*, unsigned int *count*);

Description

Language Level: XPG4, Extension

read reads *count* bytes from the file associated with *handle* into *buffer*. The read operation begins at the current position of the file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

Note: In earlier releases of VisualAge C++, read began with an underscore (_read). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _read to read for you.

Return Value

read returns the number of bytes placed in *buffer*. This number can be less than *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode. (See the note below.) The return value 0 indicates an attempt to read at end-of-file. A return value –1 indicates an error. If –1 is returned, the current file position is undefined, and errno is set to one of the following values:

Value	Meaning
EBADF	The given handle is incorrect, or the file is not open for reading, or the file is locked.
EOS2ERR	The call to the operating system was not successful.

Note: If the file was opened in text mode, the return value might not correspond to the number of bytes actually read. When text mode is in effect, the pair of characters carriage return and line feed are replaced by a single newline character.

This example opens the file sample.dat and attempts to read from it.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
   int  fh;
   char  buffer[20];

    memset(buffer, '\0', 20);
   printf("\nCreating  sample.dat.\n");
    system("echo Sample Program > sample.dat");
    if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
         perror("Unable to open sample.dat.");
        return  EXIT_FAILURE;
   }
    if (7 != read(fh, buffer, 7)) {
         perror("Unable to read from sample.dat.");
        close(fh);
        return  EXIT_FAILURE;
   }
    printf("Successfully read in the following:\n%s\n ", buffer);
   close(fh);
   return  0;

   /*************************************************************************
         The output should be:

        Creating  sample.dat.
         Successfully read in the following:
        Sample
   *************************************************************************/
}
```

"creat — Create New File" on page 100
"fread — Read Items" on page 236
"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"write — Writes from Buffer to File" on page 759

realloc — Change Reserved Storage Block Size

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void *realloc(void *ptr, size_t size);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

realloc changes the size of a previously reserved storage block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes. realloc allocates the new block from the same heap the original block was in.

If *ptr* is NULL, realloc reserves a block of storage of *size* bytes from the current thread's default heap (equivalent to calling malloc(*size*)).

If *size* is 0 and the *ptr* is not NULL, realloc frees the storage allocated to *ptr* and returns NULL.

Return Value

realloc returns a pointer to the reallocated storage block. The storage location of the block may be moved by the realloc function. Thus, the *ptr* argument to realloc is not necessarily the same as the return value.

If *size* is 0, realloc returns NULL. If there is not enough storage to expand the block to the given size, the original block is unchanged and realloc returns NULL.

The storage to which the return value points is aligned for storage of any type of object.

This example allocates storage for the prompted size of array and then uses realloc to reallocate the block to hold the new size of the array. The contents of the array are printed after each allocation.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long *array;                                /* start of the array                    */
    long *ptr;                                  /* pointer to array */
    int i;                                      /* index variable      */
    int num1,num2;                               /* number of entries of the array          */
    void print_array(long *ptr_array, int size);

    printf("Enter the size of the array\n");
   scanf("%i",  &num1);

    /* allocate num1 entries using malloc()                                                 */

    if ((array = (long *)malloc(num1 *sizeof(long))) != NULL) {
         for (ptr = array, i = 0; i < num1; ++i)                    /* assign values        */
              *ptr++ = i;
        print_array(array,  num1);
        printf("\n");
   }
    else {                                                                  /* malloc error          */
         perror("Out of storage");
        abort();
   }

    /* Change the size of the array ...                                                    */

    printf("Enter the size of the new array\n");
   scanf("%i",  &num2);
    if ((array =(long *) realloc(array, num2 *sizeof(long))) != NULL) {
         for (ptr = array+num1, i = num1; i <= num2; ++i)
              *ptr++ = i+2000;                          /* assign values to new elements          */
        print_array(array,  num2);
   }
    else {                                                                  /* realloc error        */
         perror("Out of storage");
        abort();
   }
   return  0;
```

```
    /*****************************************************************************
         The output should be similar to:

         Enter the size of the array
        2
         The array of size 2 is:
            array[ 0 ] = 0
            array[ 1 ] = 1
         Enter the size of the new array
        4
         The array of size 4 is:
            array[ 0 ] = 0
            array[ 1 ] = 1
            array[ 2 ] = 2002
            array[ 3 ] = 2003
    *****************************************************************************/
}

void print_array(long *ptr_array,int size)
{
    int  i;
     long *index = ptr_array;

     printf("The array of size %d is:\n", size);
     for (i = 0; i < size; ++i)                          /* print the array
                                                                  out                  */
         printf("    array[ %i ] = %li\n", i, ptr_array[i]);
}
```

"calloc — Reserve and Initialize Storage" on page 75
"_debug_realloc — Reallocate Memory Block" on page 132
"free — Release Storage Blocks" on page 238
"malloc — Reserve Storage Block" on page 376
"_msize — Return Number of Bytes Allocated" on page 413
"<malloc.h>" on page 769
"<stdlib.h>" on page 775

regcomp — Compile Regular Expression

Format

#include <regex.h>
int regcomp(regex_t *_preg_, const char *_pattern_, int _cflags_);

Description

Language Level: POSIX, XPG4

regcomp compiles the source regular expression pointed to by *pattern* into an executable version and stores it in the location pointed to by *preg*. You can then use regexec to compare the regular expression to other strings.

The *cflags* flag defines the attributes of the compilation process:

REG_EXTENDED
: Support extended regular expressions.

REG_NEWLINE
: Treat new-line character as a special end-of-line character; it then establishes the line boundaries matched by the and $ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the new-line character is treated like any other character.)

REG_ICASE
: Ignore case in match.

REG_NOSUB
: Ignore the number of subexpressions specified in *pattern*. When you compare a string to the compiled pattern (using regexec), the string must match the entire pattern. regexec then returns a value that indicates only if a match was found; it does not indicate at what point in the string the match begins, or what the matching string is.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, which can be interpreted differently depending on the current locale. The functions regcomp, regerror, regexec, and regfree use regular expressions in a similar way to the UNIX **awk**, **ed**, **grep**, and **egrep** commands. Regular expressions are described in more detail under "Regular Expressions" in the *Programming Guide*.

Return Value

If regcomp is successful, it returns 0. Otherwise, it returns an error code that you can use in a call to regerror, and the content of *preg* is undefined.

regcomp

This example compiles an extended regular expression.

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t  preg;
     char        *pattern = ".*(simple).*";
    int          rc;

     if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
          printf("regcomp() failed, returning nonzero (%d)\n", rc);
         exit(EXIT_FAILURE);
    }
     printf("regcomp() is sucessful.\n");
    return  0;

    /*********************************************************************************
          The output should be similar to :

          regcomp() is sucessful.
    *********************************************************************************/
}
```

"regerror — Return Error Message for Regular Expression" on page 457
"regexec — Execute Compiled Regular Expression" on page 459
"regfree — Free Memory for Regular Expression" on page 462
"Regular Expressions" in the *Programming Guide*
"<regex.h>" on page 772

regerror — Return Error Message for Regular Expression

Format

```
#include <regex.h>
size_t regerror(int errcode,   const regex_t *preg,
                char *errbuf, size_t errbuf_size);
```

Description

Language Level: POSIX, XPG4

regerror finds the description for the error code *errcode* for the regular expression *preg*. The description for *errcode* is assigned to *errbuf*. *errbuf_size* is a value that you provide that specifies the maximum message size that can be stored (the size of *errbuf*).

Regular expressions are described in detail under "Regular Expressions" in the *Programming Guide*.

The description strings for *errcode* are:

errcode	Description String
REG_NOMATCH	RE pattern not found
REG_BADPAT	Invalid regular expression
REG_ECOLLATE	Invalid collating element
REG_ECTYPE	Invalid character class
REG_EESCAPE	Last character is \
REG_ESUBREG	Invalid number in \digit
REG_EBRACK	[] imbalance
REG_EPAREN	\(\) or () imbalance
REG_EBRACE	\{ \} or { } imbalance
REG_BADBR	Invalid \{ \} range exp
REG_ERANGE	Invalid range exp endpoint
REG_ESPACE	Out of memory
REG_BADRPT	?*+ not preceded by valid RE
REG_ECHAR	Invalid multibyte character
REG_EBOL	anchor and not BOL
REG_EEOL	$ anchor and not EOL

The error descriptions are in the CPPWRERR.DLL message file. The directory for this file must be in your PATH environment variable for the messages to be found.

Return Value

regerror returns the size of the buffer needed to hold the string that describes the error condition.

regerror

This example compiles an invalid regular expression, and prints an error message using regerror.

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t  preg;
     char       *pattern = "a[missing.bracket";
    int         rc;
    char        buffer[100];

     if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
          regerror(rc, &preg, buffer, 100);
          printf("regcomp() failed with '%s'\n", buffer);
         exit(EXIT_FAILURE);
    }
    return  0;

    /*************************************************************************
          The output should be similar to :

          regcomp() failed with '[] imbalance'
    *************************************************************************/
}
```

"regcomp — Compile Regular Expression" on page 455
"regexec — Execute Compiled Regular Expression" on page 459
"regfree — Free Memory for Regular Expression" on page 462
"Regular Expressions" in the *Programming Guide*
"<regex.h>" on page 772

regexec — Execute Compiled Regular Expression

Format

```
#include <regex.h>
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

Description

Language Level: POSIX, XPG4

regexec compares the null-terminated *string* against the compiled regular expression *preg* to find a match between the two. (Regular expressions are described in "Regular Expressions" in the *Programming Guide*.)

nmatch is the number of substrings in *string* that regexec should try to match with subexpressions in *preg*. The array you supply for *pmatch* must have at least *nmatch* elements.

regexec fills in the elements of the array *pmatch* with offsets of the substrings in *string* that correspond to the parenthesized subexpressions of the original pattern given to regcomp to create *preg*. The zeroth element of the array corresponds to the entire pattern. If there are more than *nmatch* subexpressions, only the first *nmatch* – 1 are stored. If *nmatch* is 0, or if the REG_NOSUB flag was set when *preg* was created with regcomp, regexec ignores the *pmatch* argument.

The *eflags* flag defines customizable behavior of regexec:

REG_NOTBOL
: Indicates that the first character of *string* is not the beginning of line.

REG_NOTEOL
: Indicates that the first character of *string* is not the end of line.

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of *string*. The following rules determine which substrings are reported in *pmatch*:

1. If a subexpression participated in a match several times, regexec stores the offset of the last matching substring in *pmatch*.
2. If a subexpression did not match in the source *string*, regexec sets the offset shown in *pmatch* to –1.
3. If a subexpression contains subexpressions, the data in *pmatch* refers to the last such subexpression.
4. If a subexpression matches a zero-length string, the offsets in *pmatch* refer to the byte immediately following the matching string.

If the REG_NOSUB flag was set when *preg* was created by regcomp, the contents of *pmatch* are unspecified. If the REG_NEWLINE flag was not set when *preg* was created, new-line characters are allowed in *string*.

Note: If MB_CUR_MAX is specified as 2, the charmap file does not specify the DBCS characters, and a collating element (for example, [:a:]) is specified in the pattern, the DBCS characters will not match against the collating-element even if they have an equivalent weight to the collating-element.

Return Value

If a match is found, regexec returns 0. Otherwise, it returns a nonzero value indicating either no match or an error.

This example compiles an expression and matches a string against it. The first substring uses the full pattern. The second substring uses the sub-expression inside the full pattern.

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t       preg;
     char          *string = "a very simple simple simple string";
     char          *pattern = "\\(sim[a-z]le\\) \\1";
    int           rc;
     size_t        nmatch = 2;
    regmatch_t  pmatch[2];
```

```
    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s',returning %d.\n",
        string, pattern, rc);
    }
    else  {
        printf("With the whole expression, "
                "a matched substring \"%.*s\" is found at position %d to %d.\n",
                pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
                pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
                "a matched substring \"%.*s\" is found at position %d to %d.\n",
                pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
                pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return  0;

    /*********************************************************************
        The output should be similar to :

        With the whole expression, a matched substring "simple simple" is found
        at position 7 to 19.
        With the sub-expression, a matched substring "simple" is found
        at position 7 to 12.
    *********************************************************************/
}
```

"regcomp — Compile Regular Expression" on page 455
"regerror — Return Error Message for Regular Expression" on page 457
"regfree — Free Memory for Regular Expression" on page 462
"Regular Expressions" in the *Programming Guide*
"<regex.h>" on page 772

regfree — Free Memory for Regular Expression

Format

```
#include <regex.h>
void regfree(regex_t *preg);
```

Description

Language Level: POSIX, XPG4

regfree frees any memory that was allocated by regcomp to implement the regular expression *preg*. After the call to regfree, the expression defined by *preg* is no longer a compiled regular or extended expression.

Regular expressions are described in "Regular Expressions" in the *Programming Guide*.

Return Value

There is no return value.

This example compiles an extended regular expression and frees it.

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t       preg;
     char          *pattern = ".*(simple).*";
    int           rc;

     if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
          printf("regcomp() failed, returning nonzero (%d)\n", rc);
         exit(EXIT_FAILURE);
    }
    regfree(&preg);
     printf("Memory allocated for reg is freed.\n");
    return  0;

    /*********************************************************************
         The output should be similar to :

         Memory allocated for reg is freed.
    *********************************************************************/
}
```

"regcomp — Compile Regular Expression" on page 455
"regerror — Return Error Message for Regular Expression" on page 457
"regexec — Execute Compiled Regular Expression" on page 459
"Regular Expressions" in the *Programming Guide*
"<regex.h>" on page 772

remove — Delete File

Format

```
#include <stdio.h>
int remove(const char *filename);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

remove deletes the file specified by *filename*.

Note: You cannot remove a nonexistent file or a file that is open.

Return Value

remove returns 0 if it successfully deletes the file. A nonzero return value indicates an error.

This example uses remove to remove a file. It issues a message if an error occurs.

```
#include <stdio.h>

int main(void)
{
    char *FileName = "file2rm.mjq";
   FILE  *fp;

    fp = fopen(FileName, "w");
    fprintf(fp, "Hello world\n");
   fclose(fp);
    if (remove(FileName) != 0)
        perror("Could not remove file");
   else
        printf("File \"%s\" removed successfully.\n", FileName);
   return  0;

   /*************************************************************************
        The output should be:

        File "file2rm.mjq" removed successfully.
   *************************************************************************/
}
```

"fopen — Open Files" on page 218
"rename — Rename File" on page 464
"_rmtmp — Remove Temporary Files" on page 482
"unlink — Delete File" on page 688
"<stdio.h>" on page 774

rename — Rename File

Format

```
#include <stdio.h>    /* also in <io.h> */
int rename(const char *oldname, const char *newname);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

rename renames the file specified by *oldname* to the name given by *newname*. The *oldname* pointer must specify the name of an existing file. The *newname* pointer must not specify the name of an existing file. Both *oldname* and *newname* must be on the same drive; you cannot rename files across drives. You cannot rename a file with the name of an existing file. You also cannot rename an open file.

Return Value

rename returns 0 if successful. On an error, it returns a nonzero value.

This example uses rename to rename a file. It issues a message if errors occur.

```
#include <stdio.h>

int main(void)
{
    char *OldName = "oldfile.mjq";
    char *NewName = "newfile.mjq";
   FILE  *fp;

    fp = fopen(OldName, "w");
    fprintf(fp, "Hello world\n");
   fclose(fp);
    if (rename(OldName, NewName) != 0)
        perror("Could not rename file");
   else
        printf("File \"%s\" is renamed to \"%s\".\n", OldName, NewName);
   return  0;

   /*************************************************************************
        The output should be:

        File "oldfile.mjq" is renamed to "newfile.mjq".
   *************************************************************************/
}
```

"fopen — Open Files" on page 218
"remove — Delete File" on page 463
"<stdio.h>" on page 774

rewind — Adjust Current File Position

Format

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

rewind repositions the file pointer associated with *stream* to the beginning of the file. A call to rewind is the same as:

```
(void)fseek(stream, 0L, SEEK_SET);
```

except that rewind also clears the error indicator for the *stream*.

Return Value

There is no return value.

This example first opens a file myfile.dat for input and output. It writes integers to the file, uses rewind to reposition the file pointer to the beginning of the file, and then reads the data back in.

```
#include <stdio.h>

FILE *stream;
int data1,data2,data3,data4;

int main(void)
{
    data1 = 1;
    data2 = -37;

          /* Place data in the file                                          */

    stream = fopen("myfile.dat", "w+");
    fprintf(stream, "%d %d\n", data1, data2);

          /* Now read the data file                                          */

   rewind(stream);
    fscanf(stream, "%d", &data3);
    fscanf(stream, "%d", &data4);
    printf("The values read back in are: %d and %d\n", data3, data4);
   return  0;

   /*************************************************************************
       The output should be:

       The values read back in are: 1 and -37
   *************************************************************************/
}
```

"fgetpos — Get File Position" on page 203
"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"ftell — Get Current Position" on page 257

rewind

"lseek — Move File Pointer" on page 365
"_tell — Get Pointer Position" on page 619
"<stdio.h>" on page 774

rmdir — Remove Directory

Format

```
#include <direct.h>
int rmdir(char *pathname);
```

Description

Language Level: XPG4, Extension

rmdir deletes the directory specified by *pathname*. The directory must be empty, and it must not contain any files or subdirectories.

Note: In earlier releases of VisualAge C++, rmdir began with an underscore (_rmdir). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _rmdir to rmdir for you.

Return Value

rmdir returns the value 0 if the directory is successfully deleted. A return value of -1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCESS	One of the following has occurred: The given path name is not a directory. The directory is not empty. The directory is read only. The directory is the current working directory or root directory being used by a process.
ENOENT	The path name was not found.

This example deletes two directories: one in the root directory, and the other in the current working directory.

rmdir

```
#include <stdio.h>
#include <direct.h>
#include <string.h>

int main(void)
{
   char  *dir1,*dir2;

/*   Create the directory "aleng" in the root directory of the C: drive.          */

    dir1 = "c:\\aleng";
    if (0 == (mkdir(dir1)))
        printf("%s directory was created.\n", dir1);
   else
        printf("%s directory was not created.\n", dir1);

/*   Create the subdirectory "simon" in the current directory.                    */

    dir2 = "simon";
    if (0 == (mkdir(dir2)))
        printf("%s directory was created.\n", dir2);
   else
        printf("%s directory was not created.\n", dir2);

/*   Remove the directory "aleng" from the root directory of the C: drive.       */

    printf("Removing directory 'aleng' from the root directory.\n");
   if  (rmdir(dir1))
       perror(NULL);
   else
        printf("%s directory was removed.\n", dir1);

/*   Remove the subdirectory "simon" from the current directory.                  */

    printf("Removing subdirectory 'simon' from the current directory.\n");
   if  (rmdir(dir2))
       perror(NULL);
   else
        printf("%s directory was removed.\n", dir2);
   return  0;

   /*****************************************************************************
        The output should be:

        c:\aleng directory was created.
        simon directory was created.
        Removing directory 'aleng' from the root directory.
        c:\aleng directory was removed.
        Removing subdirectory 'simon' from the current directory.
        simon directory was removed.
   *****************************************************************************/
}
```


"chdir — Change Current Working Directory" on page 80
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"_getcwd — Get Path Name of Current Directory" on page 274

“mkdir — Create New Directory” on page 407
“<direct.h>” on page 762

_rmem_init — Initialize Memory Functions for Subsystem DLL

Format

```
int _rmem_init(void);
/* no header file - defined in runtime startup code */
```

Description

Language Level: Extension

_rmem_init initializes the memory functions for subsystem DLLs. Although subsystems do not require a runtime environment (and therefore do not call _CRT_init), they do require the library memory functions. For DLLs that do use a runtime environment, the memory functions are initialized with the environment by _CRT_init.

By default, all DLLs call the VisualAge for C++ _DLL_InitTerm function, which in turn calls _rmem_init for you. However, if you are writing your own subsystem _DLL_InitTerm function (for example, to perform actions other than memory initialization and termination), you must call _rmem_init from your version of _DLL_InitTerm before you can call any other library functions.

If your DLL contains C++ code, you must also call __ctordtorInit after _rmem_init to ensure that static constructors and destructors are initialized properly. __ctordtorInit is defined in the runtime startup code as:

```
void __ctordtorInit(void);
```

Return Value

If the memory functions were successfully initialized, _rmem_init returns 0. A return code of -1 indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of **stderr**.

This example shows the _DLL_InitTerm function from the VisualAge for C++ sample program for building subsystem DLLs, which calls _rmem_init to initialize the memory functions.

```
#pragma strings(readonly)
/******************************************************************************/
/* _DLL_InitTerm - Initialization/Termination function for the DLL that is    */
/*                 invoked by the loader.  When the /Ge- compile option is    */
/*                 used the linker is told that this is the function to       */
/*                 execute when the DLL is first loaded and last freed for    */
/*                 each  process.                                             */
/*                                                                            */
/* DLLREGISTER  - Called by _DLL_InitTerm for each process that loads the     */
/*                DLL.                                                        */
/*                                                                            */
/* DLLDEREGISTER- Called by _DLL_InitTerm for each process that frees the     */
/*                DLL.                                                        */
/*                                                                            */
/******************************************************************************/

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#pragma linkage ( PLUS, system )
int PLUS( int a, int b ) {
   return a + b ;
}

unsigned long _System _DLL_InitTerm ( unsigned long hModule, unsigned long ulFlag );

static unsigned long DLLREGISTER ( void );
static unsigned long DLLDEREGISTER( void );

#define SHARED_SEMAPHORE_NAME "SAMPLE05.DLL.LOCK"

/* The following data will be per-process data.  It will not be shared among  */
/* different processes.                                                       */

static HANDLE hSharedSem;                   /* Shared semaphore               */
static ULONG  ulProcessTotal;               /* Total of increments for a process  */
static DWORD  dwProcess;                    /* Process identifier             */

/* This is the global segment that is shared by every process                 */

#pragma data_seg( GLOBAL_SEG )

static ULONG ulProcessCount;                /* total number of processes      */
```

_rmem_init

```
/* _DLL_InitTerm() - called by the loader for DLL initialization/termination    */
/* This function must return a non-zero value if successful and a zero value    */
/* if unsuccessful.                                                              */

unsigned long _DLL_InitTerm ( unsigned long hModule, unsigned long ulFlag ) {

    /* If ulFlag is zero then initialization is required:                        */
    /*       If the shared memory pointer is NULL then the DLL is being loaded   */
    /*       for the first time so acquire the named shared storage for the      */
    /*       process control structures.    A linked list of process control     */
    /*       structures will be maintained.    Each time a new process loads this */
    /*       DLL, a new process control structure is created and it is inserted  */
    /*       at the end of the list by calling DLLREGISTER.                      */
    /*                                                                           */
    /* If ulFlag is 1 then termination is required:                              */
    /*       Call DLLDEREGISTER which will remove the process    control structure */
    /*       and free the shared memory block from its virtual address space.    */

  switch ( ulFlag ) {

    case  0:
            if ( !ulProcessCount ) {
              _rmem_init();
               /* Create the shared mutex semaphore                              */
               if ( !CreateMutex( NULL, FALSE, SHARED_SEMAPHORE_NAME )) {
                    printf("CreateMutex rc = %d\n", GetLastError());
                   return  0;
               }
            }

            /* Register the current process                                      */
            if ( DLLREGISTER() )
              return  0;
            break;

    case  1:
            /* Deregister the current process                                    */
            if ( DLLDEREGISTER() )
              return  0;

            _rmem_term();
            break;
```

```
    default:
            return  0;

  }

  /* Indicate success.    Non-zero means success!!!                                    */

  return  1;

}

/* DLLREGISTER - Registers the current process so that it can use this          */
/*               subsystem.    Called by _DLL_InitTerm when the DLL is first    */
/*               loaded for the current process.                                  */

static unsigned long DLLREGISTER( void ) {

  DWORD  pid;
  DWORD  rc;

  pid = GetCurrentProcessId();

  /* Open the shared mutex semaphore for this process.                            */
  if ( ! (hSharedSem = OpenMutex ( MUTEX_ALL_ACCESS | SYNCHRONIZE,    FALSE,
                                   SHARED_SEMAPHORE_NAME )) ) {
      rc = GetLastError();
      printf("OpenMutex rc = %d\n",    rc);
     return  rc;
  }

  /*Acquire the shared mutex semaphore.                                          */
  if ( WaitForSingleObject(hSharedSem, INFINITE) != WAIT_OBJECT_0) {
      rc = GetLastError();
      printf("WaitForSingleObject rc = %d\n", rc);
     return  rc;
  }

  /* Increment the count of processes registered.                               */
  ++ulProcessCount;

  /* Initialize the per-process data.                                           */
  ulProcessTotal = 0;
  dwProcess = pid;
```

_rmem_init

```
   /* Release the shared mutex semaphore.                                    */
   if ( ! ReleaseMutex( hSharedSem ) ) {
        rc = GetLastError();
        printf("ReleaseMutex rc = %d\n", rc);
       return  rc;
   }

   return  0;

}

/* DLLDEREGISTER - Deregisters the current process from this subsystem.     */
/*                 Called by _DLL_InitTerm when the DLL is freed for the    */
/*                 last time by the current process.                        */

static unsigned long DLLDEREGISTER( void ) {

   DWORD  rc;

   /* Acquire the shared mutex semaphore.                                    */
   if ( WaitForSingleObject(hSharedSem, INFINITE) != WAIT_OBJECT_0) {
        rc = GetLastError();
        printf("WaitForSingleObject rc = %d\n", rc);
       return  rc;
   }

   /* Decrement the count of processes registered.                           */
  --ulProcessCount;

   /* Tell the user that the current process has been deregistered.          */
   printf( "\nProcess %d has been deregistered.\n\n", dwProcess);

   /* Release the shared mutex semaphore.                                    */
   if ( ! ReleaseMutex( hSharedSem ) ) {
        rc = GetLastError();
        printf("ReleaseMutex rc = %d\n", rc);
       return  rc;
   }

   /* Close the shared mutex semaphore for this process.                     */
   if ( ! CloseHandle ( hSharedSem ) ) {
        rc = GetLastError();
        printf("CloseHandle rc = %d\n", rc);
       return  rc;
   }

 return 0;

}
```

Building Subsystem DLLs in the *Programming Guide*
"_CRT_init — Initialize DLL Runtime Environment" on page 103
"_CRT_term — Terminate DLL Runtime Environment" on page 107

_rmem_init

"_DLL_InitTerm — Initialize and Terminate DLL Environment" on page 144
"_rmem_term — Terminate Memory Functions for Subsystem DLL" on page 476

_rmem_term — Terminate Memory Functions for Subsystem DLL

Format

```
int _rmem_term(void);
/* no header file - defined in runtime startup code */
```

Description

Language Level: Extension

_rmem_term terminates the memory functions for subsystem DLLs. It is only needed for DLLs where the runtime library is statically linked.

By default, all DLLs call the VisualAge for C++ _DLL_InitTerm function, which in turn calls _rmem_term for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than memory initialization and termination), and your DLL statically links to the C runtime libraries, you need to call _rmem_term from your subsystem _DLL_InitTerm function. (For DLLs with a runtime environment, this termination is done by _CRT_term.)

If your DLL contains C++ code, you must also call __ctordtorTerm **before** you call _rmem_term to ensure that static constructors and destructors are terminated correctly. __ctordtorTerm is defined in the runtime startup code as:

```
void __ctordtorTerm(void);
```

Once you have called _rmem_term, you cannot call any other library functions.

If your DLL is dynamically linked, you cannot call library functions in the termination section of your _DLL_InitTerm function.

Return Value

_rmem_term returns 0 if the memory functions were successfully terminated. A return value of -1 indicates an error.

This example shows the _DLL_InitTerm function from the VisualAge for C++ sample program for building subsystem DLLs, which calls _rmem_term to terminate the library memory functions.

```
#pragma strings(readonly)
/******************************************************************************/
/* _DLL_InitTerm - Initialization/Termination function for the DLL that is    */
/*                 invoked by the loader.  When the /Ge- compile option is    */
/*                 used the linker is told that this is the function to       */
/*                 execute when the DLL is first loaded and last freed for    */
/*                 each  process.                                             */
/*                                                                            */
/* DLLREGISTER  - Called by _DLL_InitTerm for each process that loads the     */
/*                DLL.                                                        */
/*                                                                            */
/* DLLDEREGISTER- Called by _DLL_InitTerm for each process that frees the     */
/*                DLL.                                                        */
/*                                                                            */
/******************************************************************************/

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#pragma linkage ( PLUS, system )
int PLUS( int a, int b ) {
   return a + b ;
}

unsigned long _System _DLL_InitTerm ( unsigned long hModule, unsigned long ulFlag );

static unsigned long DLLREGISTER ( void );
static unsigned long DLLDEREGISTER( void );

#define SHARED_SEMAPHORE_NAME "SAMPLE05.DLL.LOCK"

/* The following data will be per-process data.  It will not be shared among  */
/* different processes.                                                       */

static HANDLE hSharedSem;                   /* Shared semaphore               */
static ULONG  ulProcessTotal;               /* Total of increments for a process  */
static DWORD  dwProcess;                    /* Process identifier             */

/* This is the global segment that is shared by every process                 */

#pragma data_seg( GLOBAL_SEG )

static ULONG ulProcessCount;                /* total number of processes      */
```

_rmem_term

```
/* _DLL_InitTerm() - called by the loader for DLL initialization/termination    */
/* This function must return a non-zero value if successful and a zero value    */
/* if unsuccessful.                                                                */

unsigned long _DLL_InitTerm ( unsigned long hModule, unsigned long ulFlag ) {

    /* If ulFlag is zero then initialization is required:                          */
    /*       If the shared memory pointer is NULL then the DLL is being loaded     */
    /*       for the first time so acquire the named shared storage for the        */
    /*       process control structures.    A linked list of process control       */
    /*       structures will be maintained.    Each time a new process loads this  */
    /*       DLL, a new process control structure is created and it is inserted    */
    /*       at the end of the list by calling DLLREGISTER.                        */
    /*                                                                             */
    /* If ulFlag is 1 then termination is required:                                */
    /*       Call DLLDEREGISTER which will remove the process    control structure */
    /*       and free the shared memory block from its virtual address space.      */

  switch ( ulFlag ) {

     case  0:
               if ( !ulProcessCount ) {
                 _rmem_init();
                  /* Create the shared mutex semaphore                            */
                  if ( !CreateMutex( NULL, FALSE, SHARED_SEMAPHORE_NAME )) {
                       printf("CreateMutex rc = %d\n", GetLastError());
                      return  0;
                  }
               }

               /* Register the current process                                    */
               if ( DLLREGISTER() )
                 return  0;
              break;

    case  1:
               /* Deregister the current process                                  */
               if ( DLLDEREGISTER() )
                 return  0;

              _rmem_term();
              break;
```

```
    default:
            return  0;

  }

  /* Indicate success.    Non-zero means success!!!                                */

  return  1;

}

/* DLLREGISTER - Registers the current process so that it can use this          */
/*               subsystem.    Called by _DLL_InitTerm when the DLL is first    */
/*               loaded for the current process.                                 */

static unsigned long DLLREGISTER( void ) {

  DWORD  pid;
  DWORD  rc;

  pid = GetCurrentProcessId();

  /* Open the shared mutex semaphore for this process.                           */
  if ( ! (hSharedSem = OpenMutex ( MUTEX_ALL_ACCESS | SYNCHRONIZE,    FALSE,
                                   SHARED_SEMAPHORE_NAME )) ) {
      rc = GetLastError();
      printf("OpenMutex rc = %d\n",    rc);
     return  rc;
  }

  /*Acquire the shared mutex semaphore.                                          */
  if ( WaitForSingleObject(hSharedSem, INFINITE) != WAIT_OBJECT_0) {
      rc = GetLastError();
      printf("WaitForSingleObject rc = %d\n", rc);
     return  rc;
  }

  /* Increment the count of processes registered.                               */
  ++ulProcessCount;

  /* Initialize the per-process data.                                           */
  ulProcessTotal = 0;
  dwProcess = pid;
```

```
   /* Release the shared mutex semaphore.                                    */
   if ( ! ReleaseMutex( hSharedSem ) ) {
        rc = GetLastError();
        printf("ReleaseMutex rc = %d\n", rc);
       return  rc;
   }

   return  0;

}

/* DLLDEREGISTER - Deregisters the current process from this subsystem.     */
/*                 Called by _DLL_InitTerm when the DLL is freed for the    */
/*                 last time by the current process.                        */

static unsigned long DLLDEREGISTER( void ) {

   DWORD  rc;

   /* Acquire the shared mutex semaphore.                                    */
   if ( WaitForSingleObject(hSharedSem, INFINITE) != WAIT_OBJECT_0) {
        rc = GetLastError();
        printf("WaitForSingleObject rc = %d\n", rc);
       return  rc;
   }

   /* Decrement the count of processes registered.                           */
  --ulProcessCount;

   /* Tell the user that the current process has been deregistered.          */
   printf( "\nProcess %d has been deregistered.\n\n", dwProcess);

   /* Release the shared mutex semaphore.                                    */
   if ( ! ReleaseMutex( hSharedSem ) ) {
        rc = GetLastError();
        printf("ReleaseMutex rc = %d\n", rc);
       return  rc;
   }

   /* Close the shared mutex semaphore for this process.                     */
   if ( ! CloseHandle ( hSharedSem ) ) {
        rc = GetLastError();
        printf("CloseHandle rc = %d\n", rc);
       return  rc;
   }

 return 0;

}
```

Building Subsystem DLLs in the *Programming Guide*
"_CRT_init — Initialize DLL Runtime Environment" on page 103
"_CRT_term — Terminate DLL Runtime Environment" on page 107

_rmem_term

"_DLL_InitTerm — Initialize and Terminate DLL Environment" on page 144
"_rmem_init — Initialize Memory Functions for Subsystem DLL" on page 470

_rmtmp — Remove Temporary Files

Format

```
#include <stdio.h>
int _rmtmp(void);
```

Description

Language Level: Extension

_rmtmp closes and removes all temporary files opened from the run-time environment from where the call to _rmtmp has been made.

Return Value

_rmtmp returns the number of temporary files deleted.

This example uses _rmtmp to remove a temporary file.

```
#include <stdio.h>

int main(void)
{
   int  num;
   FILE  *stream;

    if (NULL == (stream = tmpfile()))
        printf("Could not open new temporary file\n");
   else  {
        num = _rmtmp();
        printf("Number of temporary files removed = %d\n", num);
   }
   return  0;

   /*************************************************************************
        The output should be:

        Number of temporary files removed = 1
   *************************************************************************/
}
```

"_flushall — Write Buffers to Files" on page 215
"remove — Delete File" on page 463
"tmpfile — Create Temporary File" on page 626
"tmpnam — Produce Temporary File Name" on page 627
"unlink — Delete File" on page 688
"<stdio.h>" on page 774

_rotl - _rotr — Rotate Bits of Unsigned Integer

Format

```
#include <stdlib.h>      /* also in <builtin.h> */
unsigned int _rotl(unsigned int value, int shift);
unsigned int _rotr(unsigned int value, int shift);
```

Description

Language Level: Extension

These functions take a 4-byte integer *value* and rotate it by *shift* bits. _rotl rotates to the left, and _rotr to the right.

Note: Both _rotl and _rotr are built-in functions, which means they are implemented as inline instructions and have no backing code in the library. For this reason:

You cannot take the address of these functions.
You cannot parenthesize a call to either function. (Parentheses specify a call to the backing code for the function in the runtime library.)

Portability Consideration: Because the size of an int is only 2 bytes in 16-bit compilers, if you are migrating 16-bit programs, some parts of your programs may have to be rewritten if they use this function.

Return Value

Both functions return the rotated value. There is no error return.

This example uses _rotr and _rotl with different shift values to rotate the integer value 0x01234567:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned int val = 0X01234567;

    printf("The value of 0x%8.8lx rotated 4 bits to the left is 0x%8.8lx\n", val,
        _rotl(val,  4));
    printf("The value of 0x%8.8lx rotated 16 bits to the right is 0x%8.8lx\n",
         val, _rotr(val, 16));
   return  0;

  /*****************************************************************************
       The output should be:

       The value of 0x01234567 rotated 4 bits to the left is 0x12345670
       The value of 0x01234567 rotated 16 bits to the right is 0x45670123
  *****************************************************************************/
}
```

"_crotl – _crotr — Rotate Bits of Character Value" on page 102
"_lrotl - _lrotr — Rotate Bits of Unsigned Long Value" on page 364

_rotl – _rotr

"_llrotl - _llrotr — Rotate Bits of Unsigned Long Long Integer" on page 346
"_srotl - _srotr — Rotate Bits of Unsigned Short Value" on page 529
"swab — Swap Adjacent Bytes" on page 608
"<builtin.h>" on page 761
"<stdlib.h>" on page 775

rpmatch — Test for Yes/No Response Match

Format

```
#include <stdlib.h>
int rpmatch(const char *response);
```

Description

Language Level: POSIX, Extension

rpmatch tests whether the string pointed to by *response* matches either the affirmative or the negative response set by LC_MESSAGES category in the current locale.

Return Value

rpmatch returns:

1 If the response string matches the affirmative expression.
0 If the response string matches the negative expression.
−1 If the response string does not match either the affirmative or the negative expression.

This example asks for a reply, and checks the response.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char    *response;
   char    buffer[100];
   int     rc;

   printf("Enter  reply:\n");
    response = fgets(buffer, 100, stdin);
    rc = rpmatch(response);
    if (rc > 0)
         printf("Response was affirmative\n");
    else if (rc == 0)
                printf("Response was negative\n");
           else
                printf("Response was neither negative or affirmative\n");
   return  0;

   /*************************************************************************
         Assuming you enter : No

         The output should be :

         Response was negative
   *************************************************************************/
}
```

"setlocale — Set Locale" on page 499
"<stdlib.h>" on page 775

scanf — Read Data

Format

#include <stdio.h>
int scanf(const char *_format-string_, _argument-list_);

Description

Language Level: ANSI, ANSI 93, SAA, POSIX, XPG4, Extension

scanf reads data from the standard input stream **stdin** into the locations given by each entry in _argument-list_. Each _argument_ must be a pointer to a variable with a type that corresponds to a type specifier in _format-string_. The _format-string_ controls the interpretation of the input fields.

The _format-string_ can contain one or more of the following:

White-space characters, as specified by isspace (such as blanks and new-line characters). A white-space character causes scanf to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in _format-string_ matches any combination of white-space characters in the input.

Characters that are not white space, except for the percent sign character (%). A non-white-space character causes scanf to read, but not to store, a matching non-white-space character. If the next character in **stdin** does not match, scanf ends.

Format specifications, introduced by the percent sign (%). A format specification causes scanf to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

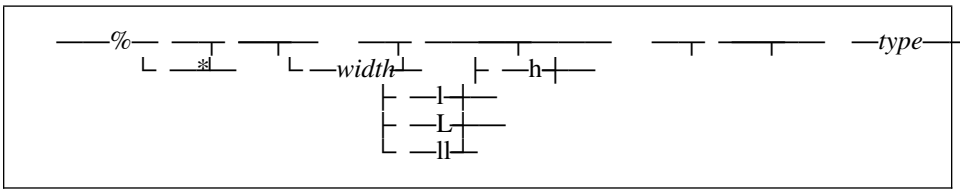
scanf reads _format-string_ from left to right. Characters outside of format specifications are expected to match the sequence of characters in **stdin**; the matched characters in **stdin** are scanned but not stored. If a character in **stdin** conflicts with _format-string_, scanf ends. The conflicting character is left in **stdin** as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first entry in _argument-list_. The second format specification converts the second input field and stores it in the second entry in _argument-list_, and so on through the end of _format-string_.

An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field _width_ is reached, whichever comes first. If there are too many arguments for the format specifications, the extra arguments are

ignored. The results are undefined if there are not enough arguments for the format specifications.

A format specification has the following form:



Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format control character, that character and following characters up to the next percent sign are treated as an ordinary sequence of characters; that is, a sequence of characters that must match the input. For example, to specify a percent-sign character, use %%.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *type*. The field is scanned but not stored.

The *width* is a positive decimal integer controlling the maximum number of characters to be read from **stdin**. No more than *width* characters are converted and stored at the corresponding *argument*. Fewer than *width* characters are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional prefix ll shows that you use the **long long** version of the following *type*, while the prefix h indicates that the **short** version is to be used. The optional prefix l, and the optional prefix L, shows that you use the **long** version of the following *type*. The corresponding *argument* should point to a **long long** object (for the ll character0, to a **long** or **double** object (for the l character), a **long double** object (for the L character), or a **short** object (with the h character). The l, ll, and h modifiers can be used with the d, i, o, x, and u *type* characters. The l modifier can also be used with the e, f, and g *type* characters. The L modifier can be used with the e, f and g *type* characters. The l and h modifiers are ignored if

specified for any other *type*. Note that the l modifier is also used with the c and s characters to indicate a multibyte character or string.

The *type* characters and their meanings are in the following table:

Character	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to **int**
o	Octal integer	Pointer to **unsigned int**
x, X	Hexadecimal integer	Pointer to **unsigned int**
i	Decimal, hexadecimal, or octal integer	Pointer to **int**
u	Unsigned decimal integer	Pointer to **unsigned int**
e, f, g E, G	Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (e or E) followed by a possibly signed integer value.	Pointer to **float**
c	Character; white-space characters that are ordinarily skipped are read when c is specified	Pointer to **char** large enough for input field
lc	Multibyte characters. The multibyte characters are converted to wide characters as if by a call to mbrtowc. The field width specifies the number of wide characters matched; if no width is specified, one character is matched.	Pointer to **wchar_t** large enough for input field
s	String	Pointer to character array large enough for input field plus a terminating null character (\0), which is automatically appended
ls	Multibyte string. None of the characters can be single-byte white-space characters (as specified by the isspace function). Each multibyte character in the sequence is converted to a wide character as if by a call to mbrtowc.	Pointer to **wchar_t** array large enough for the input field and the terminating null wide character (L\0), which is added automatically.

Character	Type of Input Expected	Type of Argument
n	No input read from *stream* or buffer	Pointer to **int**, into which is stored the number of characters successfully read from the *stream* or buffer up to that point in the call to scanf
p	Pointer to void converted to series of characters	Pointer to void

To read strings not delimited by space characters, substitute a set of characters in brackets ([ ]) for the *s* (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret ( ), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification %*a*c, where *a* is a decimal integer. In this instance, the c type character means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location, and no null character is added.

The input for a %*x* format specifier is interpreted as a hexadecimal number.

scanf scans each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on **stdin**.

Return Value

scanf returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

This example scans various types of data.

scanf

```
#include <stdio.h>

int main(void)
{
    int  i;
    float  fp;
    char  c,s[81];

     printf("Enter an integer, a real number, a character and a string : \n");
     if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
          printf("Not all of the fields were assigned\n");
    else  {
          printf("integer = %d\n", i);
          printf("real number = %f\n", fp);
          printf("character = %c\n", c);
          printf("string = %s\n", s);
    }
    return  0;

    /*************************************************************************
          The output should be similar to:

          Enter an integer, a real number, a character and a string :
          12 2.5 a yes
          integer = 12
          real number = 2.500000
          character = a
          string = yes
    *************************************************************************/
}
```

This example converts a hexadecimal integer to a decimal integer. The while loop ends if the input value is not a hexadecimal integer.

```
#include <stdio.h>

int main(void)
{
   int  number;

    printf("Enter a hexadecimal number or anything else to quit:\n");
    while (scanf("%x", &number)) {
        printf("Hexadecimal Number = %x\n", number);
        printf("Decimal Number         = %d\n", number);
   }
   return  0;

   /*************************************************************************
        The output should be similar to:

        Enter a hexadecimal number or anything else to quit:
       0x231
        Hexadecimal Number = 231
       Decimal  Number        =  561
       0xf5e
        Hexadecimal Number = f5e
       Decimal  Number        =  3934
       0x1
        Hexadecimal Number = 1
       Decimal  Number        =  1
       q
   *************************************************************************/
}
```

"_cscanf — Read Data from Keyboard" on page 112
"fscanf — Read Formatted Data" on page 245
"printf — Print Formatted Characters" on page 429
"sscanf — Read Data" on page 530
"<stdio.h>" on page 774

_searchenv — Search for File

Format

#include <stdlib.h>
void _searchenv(char *_name_, char *_env_var_, char *_path_);

Description

Language Level: Extension

_searchenv searches for the target file in the specified domain. The _env_var_ variable can be any environment variable that specifies a list of directory paths, such as PATH, LIB, INCLUDE, or other user-defined variables. Most often, it is PATH, causing a search for _name_ in all directories specified in the PATH variable.

The routine first searches for the file in the current working directory. If it does not find the file, it next looks through the directories specified by the environment variable.

If the target file is found in one of the directories, the fully-qualified file name is copied into the buffer that _path_ points to. You must ensure sufficient space for the constructed file name. If the target file is not found, _path_ contains an empty null-terminated string.

Return Value

There is no return value.

This example searches for the files _searche.c and icc.exe.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  path_buffer[_MAX_PATH];

    _searchenv("icc.exe", "PATH", path_buffer);
    printf("path: %s\n", path_buffer);
    _searchenv("_searche.c", "DPATH", path_buffer);
    printf("path: %s\n", path_buffer);
   return  0;

   /*************************************************************************
        The output should be similar to:

       path:  C:\ibmcpp\bin\icc.exe
       path:  C:\src\_searche.c
   *************************************************************************/
}
```

"getenv — Search for Environment Variables" on page 279
"putenv — Modify Environment Variables" on page 439
"<stdlib.h>" on page 775

setbuf — Control Buffering

Format

```
#include <stdio.h>
void setbuf(FILE *stream, char *buffer);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

setbuf controls buffering for the specified *stream*. The *stream* pointer must refer to an open file before any I/O or repositioning has been done.

If the *buffer* argument is NULL, the *stream* is unbuffered. If not, the *buffer* must point to a character array of length BUFSIZ, which is the buffer size defined in the <stdio.h> include file. The system uses the *buffer*, which you specify, for input/output buffering instead of the default system-allocated buffer for the given *stream*.

setbuf is similar to setvbuf, however setvbuf is more flexible and may supercede setbuf in the future.

Note: Under VisualAge for C++, streams are fully buffered by default, with the exceptions of **stderr**, which is line-buffered, and memory files, which are unbuffered.

Return Value

There is no return value.

This example opens the file setbuf.dat for writing. It then calls setbuf to establish a buffer of length BUFSIZ. When string is written to the stream, the buffer buf is used and contains the string before it is flushed to the file.

```
#include <stdio.h>
#include <string.h>

#define FILENAME   "setbuf.dat"

int main(void)
{
   char  buf[BUFSIZ];
    char string[] = "hello world";
   FILE  *stream;
```

```
    memset(buf, '\0', BUFSIZ);                  /* initialize buf to null characters    */
    stream = fopen(FILENAME, "wb");
    setbuf(stream, buf);                                                  /* set up buffer        */
    fwrite(string, sizeof(string), 1, stream);
    printf("%s\n", buf);                          /* string is found in buf now              */
    fclose(stream);                          /* buffer is flushed out to output stream */
   return  0;

   /*************************************************************************
        The output file should contain:

       hello  world

        The output should be:

       hello  world
   *************************************************************************/
}
```

"fclose — Close Stream" on page 187
"fflush — Write Buffer to File" on page 199
"_flushall — Write Buffers to Files" on page 215
"fopen — Open Files" on page 218
"setvbuf — Control Buffering" on page 508
"<stdio.h>" on page 774

_set_crt_msg_handle — Change Runtime Message Output Destination

Format

```
#include <stdio.h>
int _set_crt_msg_handle(int fh);
```

Description

Language Level: Extension

_set_crt_msg_handle changes the file handle to which runtime messages are sent, which is usually file handle 2, to *fh*. Runtime messages include exception handling messages and output from debug memory management routines.

Use _set_crt_msg_handle to trap runtime message output in applications where handle 2 is not defined, such as Windows graphical user interface applications.

The file handle *fh* must be a writable file or pipe handle. Set *fh* only for the current library environment.

Return Value

_set_crt_msg_handle returns the handle to be used for runtime message output. If an handle that is not valid is passed as an argument, it is ignored and no change takes place.

This example causes an exception by dereferencing a null pointer and uses _set_crt_msg_handle to send the exception messages to the file _scmhdl.out.

```
#include <sys\stat.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int     fh;
    char    *p = NULL;

    if (-1 == (fh = open("_scmhdl.out", O_CREAT|O_TRUNC|O_RDWR,
                                    S_IREAD|S_IWRITE)))  {
        perror("Unable to open the file _scmhdl.out.");
       exit(EXIT_FAILURE);
   }
```

```
    /* change file handle where messages are sent */
    if (fh != _set_crt_msg_handle(fh)) {
         perror("Could not change massage output handle.");
        exit(EXIT_FAILURE);
    }
    *p = 'x';                   /* cause an exception, output should be in _scmhdl.out */
    if (-1 == close(fh)) {
         perror("Unable to close _scmhdl.out.");
        exit(EXIT_FAILURE);
    }
    return  0;

    /*****************************************************************************
         Running this program would cause an exception to occur,
         the file _scmhdl.out should contain the exception messages similar to :

         Exception = c0000005 occurred at EIP = 10068.
    *****************************************************************************/
}
```

"open — Open File" on page 418
"fileno — Determine File Handle" on page 213
"_sopen — Open Shared File" on page 516
"<stdio.h>" on page 774

setjmp — Preserve Environment

Format

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

setjmp saves a stack environment that can subsequently be restored by longjmp. setjmp and longjmp provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to setjmp causes it to save the current stack environment in *env*. A subsequent call to longjmp restores the saved environment and returns control to a point corresponding to the setjmp call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when longjmp was called. The values of variables that are allocated to registers by the compiler are unpredictable. Because any auto variable could be allocated to a register in optimized code, you should consider the values of all auto variables to be unpredictable after a longjmp call.

C++ Considerations: When you call setjmp in a C++ program, ensure that the same part of the program does not also create C++ objects that need to be destroyed. When you call longjmp, objects existing at the time of the setjmp call will still exist, but any destructors called after setjmp are not called. See "longjmp — Restore Stack Environment" on page 360 for an example.

Return Value

setjmp returns the value 0 after saving the stack environment. If setjmp returns as a result of a longjmp call, it returns the *value* argument of longjmp, or 1 if the *value* argument of longjmp is 0. There is no error return value.

This example stores the stack environment at the statement

```
if(setjmp(mark) != 0) ...
```

When the system first performs the if statement, it saves the environment in *mark* and sets the condition to FALSE because setjmp returns a 0 when it saves the environment. The program prints the message

```
setjmp has been called
```

The subsequent call to function p tests for a local error condition, which can cause it to perform the longjmp function. Then, control returns to the original setjmp function using the environment saved in *mark*. This time, the condition is TRUE because -1 is the return value from the longjmp function. The program then performs the statements in the block and prints

```
longjmp has been called
```

Then the program calls the recover function and exits.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf mark;

void p(void)
{
    int error = 0;

    error = 9;

    if (error != 0)
        longjmp(mark,  -1);
}

void recover(void)
{
}

int main(void)
{
    if (setjmp(mark) != 0) {
        printf("longjmp has been called\n");
        recover();
        return  0;
    }
    printf("setjmp has been called\n");

    p();

    return  0;

    /*************************************************************************
        The output should be:

        setjmp has been called
        longjmp has been called
    *************************************************************************/
}
```

"longjmp — Restore Stack Environment" on page 360
goto in the *Language Reference*
"<setjmp.h>" on page 773

setlocale — Set Locale

Format

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

setlocale sets or queries the specified *category* of the program's *locale*. A *locale* is the complete definition of that part of a program that depends on language and cultural conventions.

The default VisualAge for C++ locale is the POSIX C locale definition, represented as "C" or "POSIX". This locale is standard for all ANSI-conforming and POSIX-conforming compilers. You can accept the default, or you can use setlocale to set the locale to one of the supplied locales listed in “Introduction to Locale” in the *Programming Guide*. Windows supports two classes of code pages: ANSI and OEM. If you want the locale that is bound to the ANSI code page, you should specify the fully qualified name that identifies the code page by name as shown in the following example.

"En_GB.IBM-1252" (English, Great Britain, code page 1252)
"Fr_CA.IBM-1252" (French, Canada, code page 1252)

If you want the locale that is bound to the OEM code page, you should specify only the language country as shown in the following example.

"En_GB.IBM-850" (English, Great Britain, code page 850 or code page 437)
"Fr_CA.IBM-850" (French, Canada, code page 850)

You may also use the fully qualified name as shown in the following example.

"En_GB" (English, Great Britain, code page 850)
"Fr_CA" (French, Canada, code page 850)

The result of setlocale depends on the arguments you specify:

To set a *category* to a specific *locale*, specify both the category and locale names. For example:

```
setlocale(LC_CTYPE, "POSIX");
```

sets the LC_CTYPE category according to the "POSIX" locale. The category names and their purpose are described in the table below.

If you specify a null string ("") for *locale*, setlocale checks locale-related environment variables in the program's environment to find a locale name or names to use for *category*. If *category* is not LC_ALL, setlocale gets the locale name from:

1. LC_ALL, if it is defined and not null
2. The environment variable with the same name as *category*, if it is defined and not null
3. LANG, if it is defined and not null
4. If none of these variables is defined to a valid locale name, setlocale uses the "C" locale.

If *category* is LC_ALL and *locale* is a null string, setlocale checks the environment variables in the same order listed above. However, it checks the variable for each *category* and sets the category to the locale specified, which could be different for each category. (By contrast, if you specified LC_ALL and a specific locale name, all categories would be set to the same locale that you specify.) The string returned lists all the locales for all the categories.

To query the locale for a *category*, specify a null pointer for *locale*. setlocale then returns a string indicating the locale setting for that *category*, without changing it. For example:

```
char *s = setlocale(LC_CTYPE, NULL);
```

returns the current locale for the LC_CTYPE category.

You can set the *category* argument of setlocale to one of these values:

Table 4 (Page 1 of 3). Locale Categories for setlocale

Category	Purpose
LC_ALL	Specifies all categories associated with the program's locale.
LC_COLLATE	Defines the collation sequence, that is, the relative order of collation elements (characters and multicharacter collation elements) in the program's locale. The collation sequence definition is used by regular expression, pattern matching, and sorting functions. Affects the regular expression functions regcomp and regexec; the string functions strcoll, strxfrm, wcscoll, and wcsxfrm. **Note:** Because both LC_CTYPE and LC_COLLATE modify the same storage area, setting LC_CTYPE and LC_COLLATE to different categories causes unpredictable results.

Table 4 (Page 2 of 3). Locale Categories for setlocale

Category	Purpose
LC_CTYPE	Defines character classification and case conversion for characters in the program's locale. Affects the behavior of character-handling functions defined in the <ctype.h> header file: csid, isalnum, isalpha, isblank, iswblank iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, iswalnum, iswalpha, iswcntrl, iswctype, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, isxdigit, tolower, toupper, towlower, towupper, wcsid, and wctype. Affects behavior of the printf and scanf families of functions: fprintf, printf, sprintf, swprintf, vfprintf, vprintf, vsprintf, vswprintf; fscanf, scanf, sscanf, and swscanf. Affects the behavior of wide character input/output functions: fgetwc, fgetws, getwc, getwchar, fputwc, fputws, putwc, putwchar, and ungetwc. Affects the behavior of multibyte and wide character conversion functions: mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, wcrtomb, wcsrtombs, wcstod, wcstol, wcstombs, wcstoul, wcswidth, wctomb, and wcwidth. **Note:** Because both LC_CTYPE and LC_COLLATE modify the same storage area, setting LC_CTYPE and LC_COLLATE to different categories causes unpredictable results.
LC_MESSAGES	Under VisualAge for C++, affects the values returned by nl_langinfo and also defines affirmative and negative response patterns, which affect rpmatch. LC_MESSAGES does **not** affect the messages for the following functions: perror, strerror, _strerror, and regerror.
LC_MONETARY	Affects monetary information returned by localeconv and strfmon. It defines the rules and symbols used to format monetary numeric information in the program's locale. The formatting rules and symbols are strings. localeconv returns pointers to these strings with names found in the <locale.h> header file.
LC_NUMERIC	Affects the decimal-point character for the formatted input/output and string conversion functions, and the nonmonetary formatting information returned by the localeconv function, specifically: printf family of functions scanf family of functions strtod atof.
LC_TIME	Defines time and date format information in the program's locale, affecting the strftime strptime, and wcsftime functions.
LC_SYNTAX	Affects the values that can be retrieved by the getsyntx function. It does **not** affect the behavior of functions that are dependent on the encoded values for formatting characters, as it may do on other platforms.

Table 4 (Page 3 of 3). Locale Categories for setlocale

Category	Purpose
LC_TOD	Affects the behavior of the functions related to time zone and Daylight Savings Time information in the program's locale. ctime, localtime, mktime, setlocale, and strftime call the tzset function to override the LC_TOD category information when TZ is defined and valid.

Return Value

setlocale returns a string that specifies the locale for the *category*. If you specified "" for *locale*, the string names the current locale; otherwise, it indicates the new locale that the *category* was set to.

If you specified LC_ALL for *category*, the returned string can be either a single locale name or, if the individual categories have different locales, a list of the locale names for each category in the following order:

1. LC_COLLATE
2. LC_CTYPE
3. LC_MONETARY
4. LC_NUMERIC
5. LC_TIME
6. LC_TOD
7. LC_MESSAGES
8. LC_SYNTAX

The string can be used on a subsequent call to restore that part of the program's locale. Because the returned string can be overwritten by subsequent calls to setlocale, you should copy the string if you plan to use it later.

If an error occurs, setlocale returns NULL and does not alter the program's locale. Errors can occur if the *category* or *locale* is not valid, or if the value of the environment variable for a category does not contain a valid locale.

This example sets the locale of the program to be "fr_fr.ibm-850" and prints the string that is associated with the locale.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#if (1 == __TOS_OS2__)
     #define LOCNAME "fr_fr.ibm-850"         /* OS/2 name              */
#else
     #define LOCNAME "fr_fr.ibm-1252"        /* Windows name           */
#endif

int main(void)
{
    char  *string;

     if (NULL == (string = setlocale(LC_ALL, LOCNAME)))
          printf("Locale \"%s\" could not be loaded\n", LOCNAME);
    else
          printf("The current locale is set to %s.\n", string);
    return  0;

    /*************************************************************************
          The OS/2 output should be similar to :

          The current locale is set to fr_fr.ibm-850.
    *************************************************************************/
}
```

This example uses setenv to set the value of the environment variable LC_TIME to FRAN, calls setlocale to set all categories to default values, then query all categories, and uses printf to print results.

setlocale

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#if (1 == __TOS_OS2__)
    #define LOCSTR "LC_TIME=fr_fr.ibm-437"        /* OS/2 name              */
#else
    #define LOCSTR "LC_TIME=fr_fr.ibm-1252"       /* Windows name           */
#endif

int main(void)
{
   char  *string;

   _putenv(LOCSTR);
    if (NULL == (string = setlocale(LC_ALL, ""))) {
        printf("setlocale(LC_ALL, \"\") failed.\n");
       exit(1);
   }  else
        printf("The current locale categories are: \"%s\"\n", string);
   return  0;

   /*************************************************************************
        The OS/2 output should be similar to :

        The current locale categories are: "C,C,C,C,fr_fr.ibm-437,C,C,C"
   *************************************************************************/
}
```

"Introduction to Locale" in the *Programming Guide*
"getenv — Search for Environment Variables" on page 279
"localeconv — Retrieve Information from the Environment" on page 352
"<locale.h>" on page 766

_setmode — Set File Translation Mode

Format

```
#include <fcntl.h>
#include <io.h>
int _setmode(int handle, int mode);
```

Description

Language Level: Extension

_setmode sets the translation mode of the file given by *handle* to *mode*. The *mode* must be one of the values in the following table:

Value	Meaning
O_TEXT	Sets the translated text mode. Carriage-return line-feed combinations are translated into a single line feed on input. Line-feed characters are translated into carriage-return line-feed combinations on output.
O_BINARY	Sets the binary (untranslated) mode. The above translations are suppressed.

Use _setmode to change the translation mode of a file handle. The translation mode only affects the read and write functions. _setmode does not affect the translation mode of streams.

If a file handle is acquired other than by a call to open, creat, _sopen or fileno, you should call _setmode for that file handle before using it within the read or write functions.

Return Value

_setmode returns the previous translation mode if successful. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EBADF	The file handle is not a handle for an open file.
EINVAL	Incorrect *mode* (neither O_TEXT nor O_BINARY)

This example uses open to create the file setmode.dat and writes to it. The program then uses _setmode to change the translation mode of setmode.dat from binary to text.

_setmode

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>

#define   FILENAME          "setmode.dat"

/* routine to validate return codes                                          */

void ckrc(int rc)
{
     if (-1 == rc) {
          printf("Unexpected return code = -1\n");
          remove(FILENAME);
          exit(EXIT_FAILURE);
     }
}

int main(void)
{
    int  h;
    int  xfer;
    int  mode;
    char  rbuf[256];
     char wbuf[] = "123\n456\n";

     ckrc(h = open(FILENAME, O_CREAT|O_RDWR|O_TRUNC|O_TEXT, S_IREAD|S_IWRITE));
     ckrc(write(h, wbuf, sizeof(wbuf)));         /* write the file (text)          */
     ckrc(lseek(h, 0, SEEK_SET));                /* seek back to the start of the file */
     ckrc(xfer = read(h, rbuf, 5));               /* read the file text             */
     printf("Read in %d characters (4 expected)\n", xfer);
     ckrc(mode = _setmode(h, O_BINARY));
     if (O_TEXT == mode)
          printf("Mode changed from binary to text\n");
    else
          printf("Previous mode was not text (unexpected)\n");
     ckrc(xfer = read(h, rbuf, 5));               /* read the file (binary)         */
     printf("Read in %d characters (5 expected)\n", xfer);
    ckrc(close(h));
    remove(FILENAME);
    return  0;

    /*****************************************************************************
          The output should be:

          Read in 4 characters (4 expected)
          Mode changed from binary to text
          Read in 5 characters (5 expected)
    *****************************************************************************/
}
```

"creat — Create New File" on page 100
"open — Open File" on page 418
"_sopen — Open Shared File" on page 516

"read — Read Into Buffer" on page 450
"write — Writes from Buffer to File" on page 759
"<fcntl.h>" on page 763
"<io.h>" on page 764
"<share.h>" on page 773
"<sys\stat.h>" on page 778

setvbuf — Control Buffering

Format

#include <stdio.h>
int setvbuf(FILE *_stream_, char *_buf_, int _type_, size_t _size_);

Description

Language Level: ANSI, SAA, XPG4

setvbuf allows control over the buffering strategy and buffer size for a specified stream. The stream must refer to a file that has been opened, but not read or written to. The array pointed to by *buf* designates an area that you provide that the C library may choose to use as a buffer for the stream. A *buf* value of NULL indicates that no such area is supplied and that the C library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

The *type* must be one of the following:

Value	Meaning
_IONBF	No buffer is used.
_IOFBF	Full buffering is used for input and output. Use *buf* as the buffer and *size* as the size of the buffer.
_IOLBF	Line buffering is used. The buffer is flushed when a new-line character is written, when the buffer is full, or when input is requested.

If *type* is _IOFBF or _IOLBF, *size* is the size of the supplied buffer. If *buf* is NULL, the C library takes *size* as the suggested size for its own buffer. If *type* is _IONBF, both *buf* and *size* are ignored.

The value for *size* must be greater than 0.

Return Value

setvbuf returns 0 if successful. It returns nonzero if an invalid value was specified in the parameter list, or if the request cannot be performed.

Note: The array used as the buffer must still exist when the specified *stream* is closed. For example, if the buffer is declared within the scope of a function block, the *stream* must be closed before the function is terminated and frees the storage allocated to the buffer.

This example sets up a buffer of buf for stream1 and specifies that input to stream2 is to be unbuffered.

```
#include <stdio.h>
#include <stdlib.h>

#define   BUF_SIZE          1024
#define   FILE1             "setvbuf1.dat"
#define   FILE2             "setvbuf2.dat"

char buf[BUF_SIZE];
FILE *stream1,*stream2;

int main(void)
{
    int flag = EXIT_SUCCESS;

    stream1 = fopen(FILE1, "r");
    stream2 = fopen(FILE2, "r");

    /* stream1 uses a user-assigned buffer of BUF_SIZE bytes                    */

    if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0) {
        printf("Incorrect type or size of buffer\n");
        flag = EXIT_FAILURE;
    }

    /* stream2 is unbuffered                                                    */

    if (setvbuf(stream2, NULL, _IONBF, 0) != 0) {
        printf("Incorrect type or size of buffer\n");
        flag = EXIT_FAILURE;
    }

   fclose(stream1);
   fclose(stream2);
   return   flag;
}
```

"fclose — Close Stream" on page 187
"fflush — Write Buffer to File" on page 199
"_flushall — Write Buffers to Files" on page 215
"fopen — Open Files" on page 218
"setbuf — Control Buffering" on page 493
"<stdio.h>" on page 774

signal — Handle Interrupt Signals

Format

#include <signal.h>
void (*signal(int *sig*, void (**sig_handler*)(int)))(int);

Description

Language Level: ANSI, SAA, XPG4, Extension

signal function assigns the signal handler *sig_handler* to handle the interrupt signal *sig*. Signals can be reported as a result of a machine interrupt (for example, division by zero) or by an explicit request to report a signal by using the raise function.

The *sig* argument must be one of the signal constants defined in <signal.h>:

Value	Meaning
SIGABRT	Abnormal termination signal sent by abort. Default action: end the program.
SIGBREAK	Ctrl-Break signal. Default action: end the program.
SIGFPE	Floating-point exceptions that are not masked, such as overflow, division by zero, and invalid operation. Default action: end the program and provide an error message. If machine-state dumps are enabled (with the /Tx compiler option), a dump is also provided.
SIGILL	Instruction not allowed. Default action: end the program and provide an error message. If machine-state dumps are enabled (with the /Tx compiler option), a dump is also provided.
SIGINT	Ctrl-C signal. Default action: end the program.
SIGSEGV	Access to memory not valid. Default action: end the program and provide an error message. If machine-state dumps are enabled (with the /Tx compiler option), a dump is also provided.
SIGTERM	Program termination signal sent by the user. Default action: end the program.
SIGUSR1	Defined by the user. Default action: ignore the signal.
SIGUSR2	Defined by the user. Default action: ignore the signal.
SIGUSR3	Defined by the user. Default action: ignore the signal.

For *sig_handler*, you must specify either the SIG_DFL or SIG_IGN constant (also defined in <signal.h>), or the address of a function that takes an integer argument (the signal).

The action taken when the interrupt signal is received depends on the value of *sig_handler*:

Value	Meaning
SIG_DFL	Perform the default action. This is the initial setting for all signals. The default actions are described in the list of signals above. All files controlled by the process are closed, but buffers are not flushed.
SIG_IGN	Ignore the interrupt signal.
sig_handler	Call the function *sig_handler*, which you provide, to handle the signal specified.

Your signal handler function (*sig_handler*) must take two integer arguments. The first argument is always the signal identifier. The second argument is 0, unless the signal is SIG_FPE. For SIG_FPE signals, the second argument passed is a floating-point error signal as defined in <float.h>. If your *sig_handler* returns, the calling process resumes running immediately following the point at which it received the interrupt signal.

After a signal is reported and the *sig_handler* is called, signal handling for that signal is reset to the default. Depending on the purpose of the signal handler, you may want to call signal inside *sig_handler* to reestablish *sig_handler* as the signal handler. You can also reset the default handling at any time by calling signal and specifying SIG_DFL.

Synchronous signals and signal handlers are not shared between threads. If you do not establish a handler for a specific signal within a thread, the default signal handling is used regardless of what handlers you may have established in other concurrent threads. However, asynchronous signals are global for the process.

Note: If an exception occurs in a math or critical library function, it is handled by the VisualAge for C++ exception handler. Your *sig_handler* will **not** be called. For more information about signals and exceptions, refer to Signal and Windows Exception Handling in the *Programming Guide*.

Return Value

All calls to signal return the address of the previous handler for the re-assigned signal.

A return value of SIG_ERR (defined in <signal.h>) indicates an error, and errno is set to EINVAL. The possible causes of the error are an incorrect *sig* value or an undefined value for *sig_handler*.

In the following example, the call to signal in main establishes the function handler to process the interrupt signal raised by abort. The handler prints a message and returns to the system.

```
#if   (1 == __TOS_OS2__)
       #define   INCL_DOSFILEMGR                    /*   For OS/2   */
       #include  <os2.h>
#else
       #include  <windows.h>                        /*   For Windows   */
#endif
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void handler(int sig)
{
#if   (1 == __TOS_OS2__)
   UCHAR  FileData[100];
   ULONG  Wrote;

    strcpy((char *)FileData, "Signal occurred.\n\r");
    DosWrite(2,   (PVOID)FileData, strlen((const char*)FileData), &Wrote);
#else
    UCHAR FileData[100] = "Signal occurred.";
   DWORD  Wrote;
    WriteFile(GetStdHandle(STD_ERROR_HANDLE), FileData, strlen(FileData),
                &Wrote,  NULL);
#endif

}

int main(void)
{
    if (SIG_ERR == signal(SIGABRT, handler)) {
        perror("Could not set SIGABRT");
       return  EXIT_FAILURE;
    }
    abort();                                  /* signal raised by abort              */
    return 0;                                 /* code should not reach here          */

   /*****************************************************************************
        The output should be:

       Signal  occurred.
   *****************************************************************************/
}
```

"abort — Stop a Program" on page 39
"atexit — Record Program Termination Function" on page 54
"exit — End Program" on page 179
"_exit — End Process" on page 180
"_fpreset — Reset Floating-Point Unit" on page 222
"raise — Send Signal" on page 448
"_spawnl - _spawnvpe — Start and Run Child Processes" on page 519
Signal and Windows Exception Handling in the *Programming Guide*

“<signal.h>” on page 773

sin — Calculate Sine

Format

#include <math.h>
double sin(double *x*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

sin calculates the sine of *x*, with *x* expressed in radians. If *x* is too large, a partial loss of significance in the result may occur.

Return Value

sin returns the value of the sine of *x*.

This example computes *y* as the sine of π/2.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  pi,x,y;

    pi = 3.1415926535;
    x = pi/2;
    y = sin(x);
    printf("sin( %lf ) = %lf\n", x, y);
   return  0;

   /*****************************************************************************
        The output should be:

        sin( 1.570796 ) = 1.000000
   *****************************************************************************/
}
```

"asin — Calculate Arcsine" on page 49
"cos — Calculate Cosine" on page 96
"_fasin — Calculate Arcsine" on page 185
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsin — Calculate Sine" on page 251
"_fsincos — Calculate Sine and Cosine" on page 252
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"<math.h>" on page 770

sinh — Calculate Hyperbolic Sine

Format

```
#include <math.h>
double sinh(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

sinh calculates the hyperbolic sine of *x*, with *x* expressed in radians.

Return Value

sinh returns the value of the hyperbolic sine of *x*. If the result is too large, sinh sets errno to ERANGE and returns the value HUGE_VAL (positive or negative, depending on the value of *x*).

This example computes *y* as the hyperbolic sine of π/2.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  pi,x,y;

    pi = 3.1415926535;
    x = pi/2;
    y = sinh(x);
    printf("sinh( %lf ) = %lf\n", x, y);
   return  0;

   /*****************************************************************************
        The output should be:

        sinh( 1.570796 ) = 2.301299
   *****************************************************************************/
}
```

"asin — Calculate Arcsine" on page 49
"cosh — Calculate Hyperbolic Cosine" on page 97
"_fasin — Calculate Arcsine" on page 185
"_fcossin — Calculate Cosine and Sine" on page 190
"_fsin — Calculate Sine" on page 251
"_fsincos — Calculate Sine and Cosine" on page 252
"sin — Calculate Sine" on page 514
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

_sopen — Open Shared File

Format

```
#include <fcntl.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>
int _sopen(char *pathname, int oflag, int shflag , int pmode);
```

Description

Language Level: Extension

_sopen opens the file specified by *pathname* and prepares the file for subsequent shared reading or writing as defined by *oflag* and *shflag*. The *oflag* is an integer expression formed by combining one or more of the constants defined in <fcntl.h>. When more than one constant is given, the constants are joined with the OR operator (|).

Oflag	Meaning
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file. This flag has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This flag applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.
O_RDWR	Open the file for both reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TRUNC	Open and truncate an existing file to 0 length. The file must have write permission. The contents of the file are destroyed. Do not specify O_TRUNC with O_RDONLY.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.
O_BINARY	Open the file in binary (untranslated) mode.
O_TEXT	Open the file in text (translated) mode. (See "Stream Processing" in the *Programming Guide* for a description of text and binary mode.)

The *shflag* argument is one of the following constants, defined in <share.h>:

Shflag	Meaning
SH_DENYRW	Deny read and write access to file.
SH_DENYWR	Deny write access to file.
SH_DENYRD	Deny read access to file.
SH_DENYNO	Permit read and write access.

There is no default value for the *shflag*.

The *pmode* argument is required only when you specify O_CREAT. If the file does not exist, *pmode* specifies the permission settings of the file, which are set when the new file is closed for the first time. If the file exists, the value of *pmode* is ignored. The *pmode* must be one of the following values, defined in <sys\stat.h>:

Value	Meaning
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If write permission is not given, the file is read-only. On the Windows operating system, all files are readable; you cannot give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE are equivalent. Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

_sopen applies the current file permission mask to *pmode* before setting the permissions. (See "umask — Sets File Mask of Current Process" on page 679 for information on file permission masks.)

Return Value

_sopen returns a file handle for the opened file. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EACCESS	The given path name is a directory, but the file is read-only and an attempt was made to open it for writing, or a sharing violation occurred.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available.
ENOENT	The file or path name was not found.
EINVAL	An incorrect argument was passed.
EOS2ERR	The call to the operating system was not successful.

This example opens the file sopen.dat for shared reading and writing using _sopen. It then opens the file for shared reading.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <share.h>

#define FILENAME "sopen.dat"

int main(void)
{
   int  fh1,fh2;

   printf("Creating  file.\n");
    system("echo Sample Program > " FILENAME);

    /* share open the file for reading and writing                              */
    if (-1 == (fh1 = _sopen(FILENAME, O_RDWR, SH_DENYNO))) {
        perror("sopen  failed");
        remove(FILENAME);
        return  EXIT_FAILURE;
    }
    /* share open the file for reading only                                     */
    if (-1 == (fh2 = _sopen(FILENAME, O_RDONLY, SH_DENYNO))) {
        perror("sopen  failed");
        close(fh1);
        remove(FILENAME);
        return  EXIT_FAILURE;
    }
    printf("File successfully opened for sharing.\n");
   close(fh1);
   close(fh2);
   remove(FILENAME);
   return  0;

   /*****************************************************************************
        The output should be:

       Creating  file.
        File successfully opened for sharing.
   *****************************************************************************/
}
```

"close — Close File Associated with Handle" on page 92
"creat — Create New File" on page 100
"open — Open File" on page 418
"fdopen — Associates Input Or Output With File" on page 194
"fopen — Open Files" on page 218
"_sopen — Open Shared File" on page 516
"umask — Sets File Mask of Current Process" on page 679
"<fcntl.h>" on page 763
"<io.h>" on page 764
"<share.h>" on page 773
"<sys\stat.h>" on page 778

_spawnl - _spawnvpe — Start and Run Child Processes

Format

```
#include <process.h>
int _spawnl(int modeflag, char *pathname, char *arg0, char *arg1, ...,
            char *argn, NULL);
int _spawnlp(int modeflag, char *pathname, char *arg0, char *arg1, ...,
            char *argn, NULL);
int _spawnle(int modeflag, char *pathname, char *arg0, char *arg1, ...,
            char *argn, NULL, char *envp[ ]);
int _spawnlpe(int modeflag, char *pathname, char *arg0, char *arg1, ...,
            char *argn, NULL, char *envp[ ]);
int _spawnv(int modeflag, char *pathname, char *argv[ ]);
int _spawnvp(int modeflag, char *pathname, char *argv[ ]);
int _spawnve(int modeflag, char *pathname, char *argv[ ], char *envp[ ]);
int _spawnvpe(int modeflag, char *pathname, char *argv[ ], char *envp[ ])
```

Description

Language Level: Extension

Each of the _spawn functions creates and runs a new child process. Enough storage must be available for loading and running the child process. All of the _spawn functions are versions of the same routine; the letters at the end determine the specific variation:

Letter	Variation
p	Uses PATH environment variable to find the file to be run
l	Lists command-line arguments separately
v	Passes to the child process an array of pointers to command-line arguments
e	Passes to the child process an array of pointers to environment strings.

The *modeflag* argument determines the action taken by the parent process before and during the _spawn. The values for *modeflag* are defined in <process.h>:

Value	Meaning
P_WAIT	Suspend the parent process until the child process is complete.
P_NOWAIT	Continue to run the parent process concurrently.
P_OVERLAY	Start the child process, and then, if successful, end the parent process. (This has the same effect as exec calls.)

The *pathname* argument specifies the file to run as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *pathname* does not have a file-name extension or end with a period, the _spawn functions add the extension .EXE and search for the file. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the _spawn functions search for *pathname* with no extension. The _spawnlp, _spawnlpe, _spawnvp, and _spawnvpe functions search for *pathname*

(using the same procedures) in the directories specified by the PATH environment variable.

You pass arguments to the child process by giving one or more pointers to character strings as arguments in the _spawn routine. These character strings form the argument list for the child process.

The argument pointers can be passed as separate arguments (_spawnl, _spawnle, _spawnlp, and _spawnlpe) or as an array of pointers (_spawnv, _spawnve, _spawnvp, and _spawnvpe). At least one argument, either arg0 or argv[0], must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. However, a different value will not produce an error.

Use the _spawnl, _spawnle, _spawnlp, and _spawnlpe functions where you know the number of arguments. The arg0 is usually a pointer to *pathname*. The arg1 through arg*n* arguments are pointers to the character strings forming the new argument list. Following arg*n*, a NULL pointer must mark the end of the argument list.

The _spawnv, _spawnve, _spawnvp, and _spawnvpe functions are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, argv. The argv[0] argument is usually a pointer to the *pathname*. The argv[1] through argv[*n*] arguments are pointers to the character strings forming the new argument list. The argv[*n*+1] argument must be a NULL pointer to mark the end of the argument list.

Files that are open when a _spawn call is made remain open in the child process. In the _spawnl, _spawnlp, _spawnv, and _spawnvp calls, the child process inherits the environment of the parent. The _spawnle, _spawnlpe, _spawnve, and _spawnvpe functions let you alter the environment for the child process by passing a list of environment settings through the envp argument. The envp argument is an array of character pointers, each element of which points to a null-terminated string, that defines an environment variable. Such a string has the form:

```
NAME=value
```

where *NAME* is the name of an environment variable, and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotation marks.) The final element of the envp array should be NULL. When envp itself is NULL, the child process inherits the environment settings of the parent process.

Note: Signal settings are not preserved in child processes created by calls to _spawn functions. The signal settings are reset to the default in the child process.

Return Value

The return from a spawn function has one of two different meanings. The return value of a synchronous spawn is the exit status of the child process. The return value of an asynchronous spawn is the process identification of the child process. You can use wait or _cwait to get the child process exit code if an asynchronous spawn was done.

A return value of -1 indicates an error (the child process is not started), and **errno** is set to one of the following values:

Value	Meaning
EAGAIN	The limit of the number of processes that the operating system permits has been reached.
EINVAL	The *modeflag* argument is incorrect.
ENOENT	The file or path name was not found or was not specified correctly.
ENOEXEC	The specified file is not executable or has an incorrect executable file format.
ENOMEM	Not enough storage is available to run the child process.

This example calls four of the eight _spawn routines. When called without arguments from the command line, the program first runs the code for case PARENT. It spawns a copy of itself, waits for its child process to run, and then spawns a second child process. The instructions for the child process are blocked to run only if argv[0] and one parameter were passed (case CHILD). In its turn, each child process spawns a grandchild as a copy of the same program. The grandchild instructions are blocked by the existence of two passed parameters. The grandchild process can overlay the child process. Each of the processes prints a message identifying itself.

```
#include   <stdio.h>
#include   <process.h>

#define    PARENT          1
#define    CHILD           2
#define    GRANDCHILD      3

int main(int argc, char **argv, char **envp)
{
    int      result;
    char     *args[4];
    switch(argc)
    {
```

```
        case PARENT:        /* no argument was passed:    spawn child and wait */
             result = spawnle(P_WAIT, argv[0], argv[0], "one", NULL, envp);
            if  (result)
                  abort();
             args[0] = argv[0];
             args[1] = "two";
             args[2] = NULL;
             /* spawn another child, and wait for it */
             result = spawnve(P_WAIT, argv[0], args, envp);
            if  (result)
                  abort();
             printf("Parent process ended\n");
            exit(0);
        case CHILD:         /* one argument passed:    allow grandchild to overlay */
             printf("child process %s began\n", argv[1]);
             if (*argv[1] == 'o')                    /* child one? */
             {
                  spawnl(P_OVERLAY, argv[0], argv[0], "one", "two", NULL);
                  abort();            /* not executed because child was overlaid */
             }
             if (*argv[1] == 't')                    /* child two? */
             {
                   args[0] = argv[0];
                   args[1] = "two";
                   args[2] = "one";
                   args[3] = NULL;

                   spawnv(P_OVERLAY, argv[0], args);

                   abort();            /* not executed because child was overlaid */
             }
             abort();            /* argument not valid */
      case GRANDCHILD:     /* two arguments passed */
           printf("grandchild %s ran\n", argv[1]);
          exit(0);
    }
/*   The output should be similar to:
      child process one began
      grandchild one ran
      child process two began
      Parent process ended
      grandchild two ran
*/
}
```

"abort — Stop a Program" on page 39
"_cwait — Wait for Child Process" on page 116
"execl - _execvpe — Load and Run Child Process" on page 175
"exit — End Program" on page 179
"_exit — End Process" on page 180
"<process.h>" on page 772

_splitpath — Decompose Path Name

Format

```
#include <stdlib.h>
void _splitpath(char *path, char *drive, char *dir,
                                char *fname, char *ext);
```

Description

Language Level: Extension

_splitpath decomposes an existing path name *path* into its four components. The *path* should point to a buffer containing the complete path name.

The maximum size necessary for each buffer is specified by the _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT constants defined in <stdlib.h>. The other arguments point to the following buffers used to store the path name elements:

Buffer	Description
drive	Contains the drive letter followed by a colon (:) if a drive is specified in *path*.
dir	Contains the path of subdirectories, if any, including the trailing slash. Slashes (/), backslashes (\), or both may be present in *path*.
fname	Contains the base file name without any extensions.
ext	Contains the file name extension, if any, including the leading period (.).

You can specify NULL for any of the buffer pointers to indicate that you do not want the string for that component returned.

The return parameters contain empty strings for any path name components not found in *path*.

Return Value There is no return value.

_splitpath

This example builds a file name path from the specified components, and then extracts the individual components.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  path_buffer[_MAX_PATH];
   char  drive[_MAX_DRIVE];
   char  dir[_MAX_DIR];
   char  fname[_MAX_FNAME];
   char  ext[_MAX_EXT];

    _makepath(path_buffer, "c", "qc\\bob\\eclibref\\e", "makepath", "c");
    printf("Path created with _makepath: %s\n\n", path_buffer);
    _splitpath(path_buffer, drive, dir, fname, ext);
    printf("Path extracted with _splitpath:\n");
    printf("drive: %s\n", drive);
    printf("directory: %s\n", dir);
    printf("file name: %s\n", fname);
    printf("extension: %s\n", ext);
   return  0;

   /*************************************************************************
        The output should be:

        Path created with _makepath: c:qc\bob\eclibref\e\makepath.c

        Path extracted with _splitpath:
       drive:  c:
       directory:  qc\bob\eclibref\e\
        file name: makepath
       extension:  .c
   *************************************************************************/
}
```

"_fullpath — Get Full Path Name of Partial Path" on page 261
"_getcwd — Get Path Name of Current Directory" on page 274
"_getdcwd — Get Full Path Name of Current Directory" on page 276
"_makepath — Create Path" on page 374
"<stdlib.h>" on page 775

sprintf — Print Formatted Data to Buffer

Format

```
#include <stdio.h>
int sprintf(char *buffer, const char *format-string, argument-list);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

sprintf formats and stores a series of characters and values in the array *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*.

The *format-string* consists of ordinary characters and has the same form and function as the *format-string* argument for the printf function. See "printf — Print Formatted Characters" on page 429 for a description of the *format-string* and arguments.

In extended mode, sprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See "Infinity and NaN Support" on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, sprintf prints (null). (In previous releases of VisualAge C++, sprintf produced no output for a null string.)

Return Value

sprintf returns the number of bytes written in the array, not counting the ending null character.

This example uses sprintf to format and print various data.

sprintf

```
#include <stdio.h>

char buffer[200];
int i,j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
     c = 'l';
     i = 35;
     fp = 1.7320508;

       /* Format and print various data                                          */

     j = sprintf(buffer, "%s\n", s);
     j += sprintf(buffer+j, "%c\n", c);
     j += sprintf(buffer+j, "%d\n", i);
     j += sprintf(buffer+j, "%f\n", fp);
     printf("string:\n%s\ncharacter count = %d\n", buffer, j);
    return  0;

   /*****************************************************************************
         The output should be:

        string:
        baltimore
        l
        35
        1.732051

         character count = 24
   *****************************************************************************/
}
```

"Infinity and NaN Support" on page 27
"_cprintf — Print Characters to Screen" on page 98
"fprintf — Write Formatted Data to a Stream" on page 224
"printf — Print Formatted Characters" on page 429
"sscanf — Read Data" on page 530
"swprintf — Format and Write Wide Characters to Buffer" on page 609
"vfprintf — Print Argument Data to Stream" on page 698
"vprintf — Print Argument Data" on page 700
"vsprintf — Print Argument Data to Buffer" on page 702
"vswprintf — Format and Write Wide Characters to Buffer" on page 704
"<stdio.h>" on page 774

sqrt — Calculate Square Root

Format

#include <math.h>
double sqrt(double *x*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

sqrt calculates the nonnegative value of the square root of *x*.

Return Value

sqrt returns the square root result. If *x* is negative, the function sets errno to EDOM, and returns 0.

This example computes the square root of the quantity passed as the first argument to main. It prints an error message if you pass a negative value.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv)
{
    double value = 45.0;

    printf("sqrt( %f ) = %f\n", value, sqrt(value));
   return  0;

   /*****************************************************************************
        The output should be:

        sqrt( 45.000000 ) = 6.708204
   *****************************************************************************/
}
```

"exp — Calculate Exponential Function" on page 181
"_fsqrt — Calculate Square Root" on page 254
"hypot — Calculate Hypotenuse" on page 301
"log — Calculate Natural Logarithm" on page 358
"log10 — Calculate Base 10 Logarithm" on page 359
"pow — Compute Power" on page 428
"<math.h>" on page 770

srand — Set Seed for rand Function

Format

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

srand sets the starting point for producing a series of pseudo-random integers. If srand is not called, the rand seed is set as if srand(1) were called at program start. Any other value for *seed* sets the generator to a different starting point.

The rand function generates the pseudo-random numbers.

Return Value

There is no return value.

This example first calls srand with a value other than 1 to initiate the random value sequence. Then the program computes five pseudo-random values for the array of integers called ranvals.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int  i,ranvals[5];

   srand(17);
    for (i = 0; i < 5; i++) {
        ranvals[i] = rand();
        printf("Iteration %d ranvals [%d] = %d\n", i+1, i, ranvals[i]);
   }
   return  0;

   /*************************************************************************
        The output should be similar to:

        Iteration 1 ranvals [0] = 24107
        Iteration 2 ranvals [1] = 16552
        Iteration 3 ranvals [2] = 12125
        Iteration 4 ranvals [3] = 9427
        Iteration 5 ranvals [4] = 13152
   *************************************************************************/
}
```

"rand — Generate Random Number" on page 449
"<stdlib.h>" on page 775

_srotl - _srotr — Rotate Bits of Unsigned Short Value

```
#include <stdlib.h>     /* also in <builtin.h> */

unsigned short _srotl(unsigned short value, int shift);
unsigned short _srotr(unsigned short value, int shift);
```

Description

Language Level: Extension

The _srotl and _srotr functions rotate the unsigned short integer *value* by *shift* bits. The _srotl function rotates to the left, and _srotr to the right.

Note: Both _srotl and _srotr are built-in functions, which means they are implemented as inline instructions and have no backing code in the library. For this reason:

You cannot take the address of these functions.
You cannot parenthesize a call to either function. (Parentheses specify a call to the function's backing code, and these functions have none.)

Return Value

Both functions return the rotated value. There is no error return value.

This example uses _srotl and _srotr with different shift values to rotate the character value:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned short val = 0X0123;

    printf("The value of 0x%4.4x rotated 4 bits to the left is 0x%4.4x\n", val,
        _srotl(val,  4));
    printf("The value of 0x%4.4x rotated 8 bits to the right is 0x%4.4x\n",
         val, _srotr(val, 8));
   return  0;

   /*****************************************************************************
        The output should be:

        The value of 0x0123 rotated 4 bits to the left is 0x1230
        The value of 0x0123 rotated 8 bits to the right is 0x2301
   *****************************************************************************/
}
```

"_crotl – _crotr — Rotate Bits of Character Value" on page 102
"_lrotl - _lrotr — Rotate Bits of Unsigned Long Value" on page 364
"_rotl - _rotr — Rotate Bits of Unsigned Integer" on page 483
"<stdlib.h>" on page 775
"<builtin.h>" on page 761

sscanf — Read Data

Format

#include <stdio.h>
int sscanf(const char *_buffer_, const char *_format_, _argument-list_);

Description

Language Level: ANSI, SAA, POSIX, XPG4, Extension

sscanf reads data from _buffer_ into the locations given by _argument-list_. Each _argument_ must be a pointer to a variable with a type that corresponds to a type specifier in the _format-string_. See "scanf — Read Data" on page 486 for a description of the _format-string_.

Return Value

sscanf returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF when the end of the string is encountered before anything is converted.

This example uses sscanf to read various data from the string _tokenstring_, and then displays that data.

```
#include <stdio.h>

#define   SIZE              81

char *tokenstring = "15 12 14";
int i;
float fp;
char s[SIZE];
char c;

int main(void)
{

     /* Input various data                                                    */

    sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
```

```
    /* If there were no space between %s and %c,                    */
    /* sscanf would read the first character following             */
    /* the string, which is a blank space.                          */
    /* Display the data                                                */

   printf("string = %s\n", s);
   printf("character = %c\n", c);
   printf("integer = %d\n", i);
   printf("floating-point number = %f\n", fp);
  return  0;

  /*****************************************************************************
      The output should be:

      string = 15
      character = 1
      integer = 2
      floating-point number = 14.000000
  *****************************************************************************/
}


/*****************  Output should be similar to:  ***************** */
```

"Infinity and NaN Support" on page 27
"_cscanf — Read Data from Keyboard" on page 112
"fscanf — Read Formatted Data" on page 245
"scanf — Read Data" on page 486
"sprintf — Print Formatted Data to Buffer" on page 525
"swscanf — Read Wide Characters from Buffer" on page 611
"<stdio.h>" on page 774

stat — Get Information about File or Directory

Format

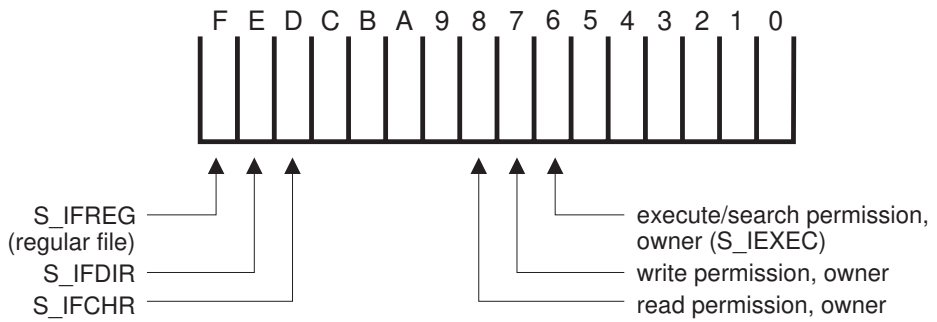
```
#include <sys\types.h>
#include <sys\stat.h>
int stat(const char *pathname, struct stat *buffer)      ;
```

Description

Language Level: XPG4, Extension

stat stores information about the file or directory specified by *pathname* in the structure to which *buffer* points. The stat structure, defined in <sys\stat.h>, contains the following fields:

Field	Value
st_mode	Bit mask for file-mode information. The S_IFDIR bit is set if *pathname* specifies a directory. The S_IFREG bit is set if *pathname* specifies an ordinary file. User read/write bits are set according to the permission mode of the file. The S_IEXEC bit is set using the file name extension. The other bits are undefined.



st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.
st_size	Size of the file in bytes.
st_atime	Time of last access of file.
st_mtime	Time of last modification of file.
st_ctime	Time of file creation.

Note: If the given *pathname* specifies only a device (for example C:\), time information is not available.

Note: In earlier releases of VisualAge C++, stat began with an underscore (_stat). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _stat to stat for you.

Return Value

stat returns the value 0 if the file status information is obtained. A return value of -1 indicates an error, and **errno** is set to ENOENT, indicating that the file name or path name could not be found.

This example requests that the status information for the file test.exe be placed into the structure *buf*. If the request is successful and the file is executable, the example runs test.exe.

```
#include <sys\types.h>
#include <sys\stat.h>
#include <process.h>
#include <stdio.h>

int main(void)
{
    struct stat buf;

    if (0 == stat("test.exe", &buf)) {
        if ((buf.st_mode&S_IFREG) && (buf.st_mode&S_IEXEC))
            execl("test.exe", "test", NULL);          /* file is executable      */
    }
    else
        printf("File could not be found\n");
    return  0;

    /*************************************************************************
        The source for test.exe is:

        #include  <stdio.h>
        int  main(void)
        {
            puts("test.exe is an executable file");
        }

        The output should be:

        test.exe is an executable file
    *************************************************************************/
}
```

"fstat — Information about Open File" on page 255
"<sys\stat.h>" on page 778
"<sys\types.h>" on page 778

_status87 — Get Floating-Point Status Word

Format

```
#include <float.h>    /* also in <builtin.h> */
unsigned int _status87(void);
```

Description

Language Level: Extension

_status87 gets the current floating-point status word. The floating-point status word is a combination of the numeric coprocessor status word and other conditions detected by the numeric exception handler, such as floating-point stack underflow and overflow.

Return Value

The bits in the value returned reflect the floating-point status for the current thread only before the call was made. These bits are defined in the <float.h> include file. _status87 does not affect any other threads that may be processing.

This example uses _status87 to get the value of the floating-point status word.

```
#include <stdio.h>
#include <float.h>

double a = 1e-40,b;
float x,y;

int main(void)
{
    printf("status = 0x%.4x - clear\n", _status87());

    /* change control word to mask all exceptions                    */

    _control87(0x037f,  0xffff);
    y = a;                          /* store into y is inexact and causes underflow */
    printf("status = 0x%.4X - inexact, underflow\n", _status87());
```

```
    /* reinitialize the floating point unit                                    */

    _fpreset();

    /* change control word to mask all exceptions                              */

    _control87(0x037f,  0xffff);
     b = y;                                                  /* y is denormal  */
     printf("status = 0x%.4X - denormal\n", _status87());

    /* reinitialize the floating point unit                                    */

    _fpreset();
    return  0;

    /*************************************************************************
         The output should be:

         status = 0x0000 - clear
         status = 0x0030 - inexact, underflow
         status = 0x0002 - denormal
    *************************************************************************/
}
```

"_clear87 — Clear Floating-Point Status Word" on page 88
"_control87 — Set Floating-Point Control Word" on page 94
"_fpreset — Reset Floating-Point Unit" on page 222
"<float.h>" on page 763

strcat — Concatenate Strings

Format

#include <string.h>
char *strcat(char *_string1_, const char *_string2_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strcat concatenates _string2_ to _string1_ and ends the resulting string with the null character.

strcat operates on null-terminated strings. The string arguments to the function should contain a null character (\0) marking the end of the string. No length checking is performed. You should not use a literal string for a _string1_ value, although _string2_ may be a literal string.

If the storage of _string1_ overlaps the storage of _string2_, the behavior is undefined.

Return Value

strcat returns a pointer to the concatenated string (_string1_).

This example creates the string "computer program" using strcat.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char buffer1[SIZE] = "computer";
   char  *ptr;

    ptr = strcat(buffer1, " program");
    printf("buffer1 = %s\n", buffer1);
   return  0;

   /*****************************************************************************
        The output should be:

        buffer1 = computer program
   *****************************************************************************/
}
```

“strchr — Search for Character” on page 537
“strcmp — Compare Strings” on page 538
“strcpy — Copy Strings” on page 544
“strcspn — Compare Strings for Substrings” on page 546
“strncat — Concatenate Strings” on page 566
“wcscat — Concatenate Wide-Character Strings” on page 708
“wcsncat — Concatenate Wide-Character Strings” on page 723
“<string.h>” on page 777

strchr — Search for Character

Format

```
#include <string.h>
char *strchr(const char *string, int c);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strchr finds the first occurrence of a character in a string. The character *c* can be the null character (\0); the ending null character of *string* is included in the search.

The strchr function operates on null-terminated strings. The string arguments to the function should contain a null character (\0) marking the end of the string.

Return Value

strchr returns a pointer to the first occurrence of *c* converted to a character in *string*. The function returns NULL if the specified character is not found.

This example finds the first occurrence of the character p in "computer program".

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char buffer1[SIZE] = "computer program";
   char  *ptr;
    int ch = 'p';

    ptr = strchr(buffer1, ch);
    printf("The first occurrence of %c in '%s' is '%s'\n", ch, buffer1, ptr);
   return  0;

   /*****************************************************************************
       The output should be:

       The first occurrence of p in 'computer program' is 'puter program'
   *****************************************************************************/
}
```

"strcmp — Compare Strings" on page 538
"strcspn — Compare Strings for Substrings" on page 546
"strncmp — Compare Strings" on page 568
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcsspn — Search Wide-Character Strings" on page 735
"<string.h>" on page 777

strcmp — Compare Strings

Format

#include <string.h>
int strcmp(const char *_string1_, const char *_string2_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strcmp compares _string1_ and _string2_. The function operates on null-terminated strings. The string arguments to the function should contain a null character (\0) marking the end of the string.

Return Value

strcmp returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	_string1_ less than _string2_
0	_string1_ identical to _string2_
Greater than 0	_string1_ greater than _string2_.

If _count_ is greater than the length of _string1_ or _string2_, characters that follow a null character are not compared.

This example compares the two strings passed to main using strcmp.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   int  result;

    if (argc != 3) {
        printf("Usage: %s string1 string2\n", argv[0]);
   }
   else  {
        result = strcmp(argv[1], argv[2]);
        if (0 == result)
            printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
       else
            if (result < 0)
                printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
           else
                printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
   }
   return  0;

   /*****************************************************************************
        If the following arguments are passed to this program:

        "is this first?" "is this before that one?"

        The output should be:

        "is this first?" is greater than "is this before that one?"
   *****************************************************************************/
}
```

"strcmpi — Compare Strings Without Case Sensitivity" on page 540
"strcoll — Compare Strings Using Collation Rules" on page 542
"strcspn — Compare Strings for Substrings" on page 546
"stricmp — Compare Strings as Lowercase" on page 562
"strncmp — Compare Strings" on page 568
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"wcscmp — Compare Wide-Character Strings" on page 711
"wcsncmp — Compare Wide-Character Strings" on page 725
"<string.h>" on page 777

strcmpi — Compare Strings Without Case Sensitivity

Format

```
#include <string.h>
int strcmpi(const char *string1, const char *string2);
```

Description

Language Level: Extension

strcmpi compares *string1* and *string2* without sensitivity to case. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

Return Value

strcmpi returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* equivalent to *string2*
Greater than 0	*string1* greater than *string2*.

If *n* is greater than the length of *string1* or *string2*, characters that follow a *null* character are *not* compared.

This example uses strcmpi to compare two strings.

```
#include <stdio.h>
#include <string.h>

int main(void)
{

    /* Compare two strings without regard to case                          */

    if (0 == strcmpi("hello", "HELLO"))
        printf("The strings are equivalent.\n");
    else
        printf("The strings are not equivalent.\n");
    return  0;

    /*************************************************************************
        The output should be:

        The strings are equivalent.
    *************************************************************************/
}
```

"strcoll — Compare Strings Using Collation Rules" on page 542
"strcspn — Compare Strings for Substrings" on page 546
"strdup — Duplicate String" on page 549

"stricmp — Compare Strings as Lowercase" on page 562
"strncmp — Compare Strings" on page 568
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"wcscmp — Compare Wide-Character Strings" on page 711
"wcsncmp — Compare Wide-Character Strings" on page 725
"<string.h>" on page 777

strcoll — Compare Strings Using Collation Rules

Format

```
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

Description

Language Level: ANSI, SAA, XPG4

strcoll compares the string pointed to by *string1* against the string pointed to by *string2*, both interpreted according to the information in the LC_COLLATE category of the current locale.

strcoll differs from the strcmp function. strcoll performs a comparison between two character strings based on language collation rules as controlled by the LC_COLLATE category. In contrast, strcmp performs a character to character comparison.

Return Value

strcoll returns an integer value indicating the relationship between the strings, as listed below:

Value	Meaning
Less than 0	*string1* is less than *string2*
0	*string1* is equivalent to *string2*
Greater than 0	*string1* is greater than *string2*

Notes:

1. strcoll might need to allocate additional memory to perform the comparison algorithm specified in the LC_COLLATE. If the memory request cannot be satisfied (by malloc), then strcoll fails.
2. If the locale supports double-byte characters (MB_CUR_MAX specified as 2), strcoll validates the multibyte characters. (Previously, strcoll did not validate the string.) strcoll will fail if the string contains invalid multibyte characters.
3. If MB_CUR_MAX is specified as 2, but the charmap file does not specify the DBCS characters, the DBCS characters will collate after the single-byte characters.

This example compares the two strings passed to *main*.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   int  result;

    if (argc != 3) {
        printf("Usage: %s string1 string2\n", argv[0]);
   }
   else  {
        result = strcoll(argv[1], argv[2]);
        if (0 == result)
            printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
       else
            if (result < 0)
                printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
           else
                printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
   }
   return  0;

   /*************************************************************************
        If the program is passed the following arguments:

       "firststring"  "secondstring"

        The output should be:

        "firststring" is less than "secondstring"
   *************************************************************************/
}
```

“setlocale — Set Locale” on page 499
“strcmp — Compare Strings” on page 538
“strcmpi — Compare Strings Without Case Sensitivity” on page 540
“strncmp — Compare Strings” on page 568
“wcscoll — Compare Wide-Character Strings” on page 713
“<string.h>” on page 777

strcpy — Copy Strings

Format

#include <string.h>
char *strcpy(char **string1*, const char **string2*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strcpy copies *string2*, including the ending null character, to the location specified by *string1*.

strcpy operates on null-terminated strings. The string arguments to the function should contain a null character (\0) marking the end of the string. No length checking is performed. You should not use a literal string for a *string1* value, although *string2* may be a literal string.

Return Value

strcpy returns a pointer to the copied string (*string1*.).

This example copies the contents of source to destination.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char source[SIZE] = "123456789";
    char source1[SIZE] = "123456789";
    char destination[SIZE] = "abcdefg";
    char destination1[SIZE] = "abcdefg";
   char  *return_string;
    int index = 5;

    /* This is how strcpy works                                          */

    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("After strcpy, destination becomes '%s'\n\n", destination);

    /* This is how strncpy works                                         */

    printf("destination1 is originally = '%s'\n", destination1);
    return_string = strncpy(destination1, source1, index);
    printf("After strncpy, destination1 becomes '%s'\n", destination1);
   return  0;
```

```
/*****************************************************************************
     The output should be:

     destination is originally = 'abcdefg'
     After strcpy, destination becomes '123456789'

     destination1 is originally = 'abcdefg'
     After strncpy, destination1 becomes '12345fg'
 *****************************************************************************/
}
```

"strcat — Concatenate Strings" on page 536
"strdup — Duplicate String" on page 549
"strncpy — Copy Strings" on page 570
"wcscpy — Copy Wide-Character Strings" on page 715
"wcsncpy — Copy Wide-Character Strings" on page 727
"<string.h>" on page 777

strcspn — Compare Strings for Substrings

Format

#include <string.h>
size_t strcspn(const char *_string1_, const char *_string2_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strcspn finds the first occurrence of a character in _string1_ that belongs to the set of characters specified by _string2_. Ending null characters are not considered in the search.

The strcspn function operates on null-terminated strings. The string arguments to the function should contain a null character (\0) marking the end of the string.

Return Value

strcspn returns the index of the first character found. This value is equivalent to the length of the initial substring of _string1_ that consists entirely of characters not in _string2_.

This example uses strcspn to find the first occurrence of any of the characters a, x, l or e in string.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char string[SIZE] = "This is the source string";
    char *substring = "axle";

    printf("The first %i characters in the string \"%s\" are not in the "
        "string \"%s\" \n", strcspn(string, substring), string, substring);
   return  0;

   /*****************************************************************************
        The output should be:

        The first 10 characters in the string "This is the source string" are not
        in the string "axle"
   *****************************************************************************/
}
```

“strchr — Search for Character” on page 537
“strcmp — Compare Strings” on page 538
“strcmpi — Compare Strings Without Case Sensitivity” on page 540
“stricmp — Compare Strings as Lowercase” on page 562
“strncmp — Compare Strings” on page 568
“strnicmp — Compare Strings Without Case Sensitivity” on page 572
“strpbrk — Find Characters in String” on page 575

“strspn — Search Strings” on page 585
“wcscmp — Compare Wide-Character Strings” on page 711
“wcsncmp — Compare Wide-Character Strings” on page 725
“<string.h>” on page 777

_strdate — Copy Current Date

Format

```
#include <time.h>
char *_strdate(char *date);
```

Description

Language Level: Extension

_strdate stores the current date as a string in the buffer pointed to by *date* in the following format:

```
mm/dd/yy
```

The two digits *mm* represent the month, the digits *dd* represent the day of the month, and the digits *yy* represent the year. For example, the string 10/08/91 represents October 8, 1991. The buffer must be at least 9 bytes.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Return Value

_strdate returns a pointer to the buffer containing the date string. There is no error return.

This example prints the current date.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char  buffer[9];

    printf("The current date is %s \n", _strdate(buffer));
   return  0;

   /*************************************************************************
       The output should be similar to:

       The current date is 01/02/95
   *************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"_ftime — Store Current Time" on page 259
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"time — Determine Current Time" on page 625
"tzset — Assign Values to Locale Information" on page 634
"<time.h>" on page 779

strdup — Duplicate String

Format

```
#include <string.h>
char *strdup(const char *string);
```

Description

Language Level: XPG4, Extension

strdup reserves storage space for a copy of *string* by calling malloc. The string argument to this function is expected to contain a null character (\0) marking the end of the string. Remember to free the storage reserved with the call to strdup.

Return Value

strdup returns a pointer to the storage space containing the copied string. If it cannot reserve storage strdup returns NULL.

This example uses strdup to duplicate a string and print the copy.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string = "this is a copy";
   char  *newstr;

    /* Make newstr point to a duplicate of string                                    */

    if ((newstr = strdup(string)) != NULL)
        printf("The new string is: %s\n", newstr);
   return  0;

   /*****************************************************************************
        The output should be:

        The new string is: this is a copy
   *****************************************************************************/
}
```

"strcpy — Copy Strings" on page 544
"strncpy — Copy Strings" on page 570
"wcscpy — Copy Wide-Character Strings" on page 715
"wcsncpy — Copy Wide-Character Strings" on page 727
"<string.h>" on page 777

strerror — Set Pointer to Runtime Error Message

Format

#include <string.h>
char *strerror(int *errnum*);

Description

Language Level: ANSI, SAA, XPG4

strerror maps the error number in *errnum* to an error message string.

Return Value

strerror returns a pointer to the string. It does not use the program locale in any way.

The value of errno may be set to:

EILSEQ An encoding error has occurred converting a multibyte character.
E2BIG The output buffer is too small.

This example opens a file and prints a runtime error message if an error occurs.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define FILENAME "strerror.dat"

int main(void)
{
   FILE  *stream;

    if (NULL == (stream = fopen(FILENAME, "r")))
        printf(" %s \n", strerror(errno));
   return  0;
}
```

"clearerr — Reset Error Indicators" on page 87
"ferror — Test for Read/Write Errors" on page 198
"perror — Print Error Message" on page 427
"_strerror — Set Pointer to System Error String" on page 551
"<string.h>" on page 777

_strerror — Set Pointer to System Error String

Format

```
#include <string.h>
char *_strerror(char *string);
```

Description

Language Level: Extension

_strerror tests for system error. It gets the system error message for the last library call that produced an error and prefaces it with your *string* message.

Your *string* message can be a maximum of 94 bytes.

Unlike perror, _strerror by itself does not print a message. To print the message returned by _strerror to **stdout**, use a printf statement similar to the following:

```
if ((access("datafile",2)) == -1)
   printf(stderr,_strerror(NULL));
```

You could also print the message with an fprintf statement.

To produce accurate results, call _strerror immediately after a library function returns with an error. Otherwise, subsequent calls might write over the **errno** value.

Return Value

If *string* is equal to NULL, _strerror returns a pointer to a string containing the system error message for the last library call that produced an error, ended by a new-line character (\n).

If *string* is not equal to NULL, _strerror returns a pointer to a string containing:

Your string message
A colon
A space
The system error message for the last library call producing an error
A new-line character (\n).

This example shows how _strerror can be used with the fopen function.

```
#include <string.h>
#include <stdio.h>

#define INFILE       "_strerro.in"
#define OUTFILE      "_strerro.out"

int main(void)
{
   FILE  *fh1,*fh2;

    fh1 = fopen(INFILE, "r");
    if (NULL == fh1)
       /*    the error message goes through stdout not stderr                    */
         printf(_strerror("Open failed on input file"));
    fh2 = fopen(OUTFILE, "w+");
    if (NULL == fh2)
         printf(_strerror("Open failed on output file"));
   else
         printf("Open on output file was successful.\n");
    if (fh1 != NULL)
        fclose(fh1);
    if (fh2 != NULL)
        fclose(fh2);
   remove(OUTFILE);
   return  0;

   /*****************************************************************************
         The output should be:

         Open failed on input file: The file cannot be found.
         Open on output file was successful.
   *****************************************************************************/
}
```

"clearerr — Reset Error Indicators" on page 87
"ferror — Test for Read/Write Errors" on page 198
"perror — Print Error Message" on page 427
"strerror — Set Pointer to Runtime Error Message" on page 550
"<string.h>" on page 777

strfmon — Convert Monetary Value to String

Format

#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, ...);

Description

Language Level: XPG4

strfmon places characters into the array pointed to by *s*, as controlled by the string pointed to by *format*. No more than *maxsize* characters are placed into the array.

The character string *format* contains two types of objects:

Plain characters, which are copied to the output array.
Directives, each of which results in the fetching of zero or more arguments that are converted and formatted.

The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are simply ignored. If objects pointed to by *s* and *format* overlap, the behavior is undefined.

The directive (conversion specification) consists of the following sequence.

1. A % character
2. Optional flags, described below: =*f*, , then +, C, (, then !
3. Optional field width (may be preceded by -)
4. Optional left precision: #*n*
5. Optional right precision: .*p*
6. Required conversion character to indicate what conversion should be performed: i or n.

Each directive is replaced by the appropriate characters, as described in the following list:

%i The double argument is formatted according to the locale's international currency format (for example, in USA: USD 1,234.56).

%n The double argument is formatted according to the locale's national currency format (for example, in USA: $1,234.56).

%% is replaced by %. No argument is converted.

You can give optional conversion specifications immediately after the initial % of a directive in the following order:

Specifier	Meaning
=f	Specifies *f* as the numeric fill character. This flag is used in conjunction with the maximum digits specification *#n* (see below). The default numeric fill character is the space character. This option does not affect the other fill operations that always use a space as the fill character.
	Formats the currency amount without thousands grouping characters. The default is to insert the grouping characters if defined for the current locale.
+ | C | (	Specifies the style of representing positive and negative currency amounts. You can specify only one of +, C, or (. The + specifies to use the locale's equivalent of + and –. C specifies to use the locale's equivalent of DB for negative and CR for positive. The (specifies to use the locale's equivalent of enclosing negative amounts within parenthesis. If this option is not included, a default specified by the current locale is used.
!	Suppresses the currency symbol from the output conversion.
[–]w	A decimal digit string that specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the – flag is specified).
#n	A decimal digit string that specifies a maximum number of digits expected to be formatted to the left of the radix character. You can use this option to keep the formatted output from multiple calls to strfmon aligned in the same columns. You can also use it to fill unused positions with a special character, as in $***123.45. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more digit positions are required than specified, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the *=f* specification above). If thousands grouping is enabled, the behavior is: 1. Format the number as if it is an *n* digit number. 2. Insert fill characters to the left of the leftmost digit (for example, $0001234.56 or $***1234.56). 3. Insert the separator character (for example, $0,001,234.56 or $*,**1,234.56). 4. If the fill character is not the digit zero, the separators are replaced by the fill character (for example, $****1,234.56).

To ensure alignment, any characters appearing before or after the number in the formatted output, such as currency or sign symbols, are padded with space characters to make their positive and negative formats an equal length.

.p A decimal digit string that specifies the number of digits after the radix character. If the value of the precision *p* is 0, no radix character appears. If this option is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

The LC_MONETARY category of the program's locale affects the behavior of this function, including the monetary radix character (which is different from the numeric radix character affected by the LC_NUMERIC category), the thousands (or alternate grouping) separator, the currency symbols, and formats. The international currency symbol should be in accordance with those specified in ISO 4217 Codes for the representation of currencies and funds.

Return Value If the total number of resulting bytes including the terminating null character is not more than *maxsize*, strfmon returns the number of bytes placed into the array pointed to by *s*, not including the terminating null character. Otherwise, strfmon returns -1 and the contents of the array are indeterminate.

This example uses strfmon to format the monetary value for money, then prints the resulting string.

strfmon

```
#include <monetary.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "en_us.ibm-437"        /* OS/2 name              */
#else
    #define LOCNAME "en_us.ibm-1252"       /* Windows name           */
#endif

int main(void)
{
   char    string[100];        /* hold the string returned from strfmon() */
    double money = 1234.56;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
    }
    strfmon(string, 100, "%i", money);
    printf("International currency format = \"%s\"\n", string);
    strfmon(string, 100, "%n", money);
    printf("National currency format          = \"%s\"\n", string);
   return  0;

   /*************************************************************************
        The output should be similar to :

        International currency format = "USD 1,234.56"
        National currency format          = "$1,234.56"
   *************************************************************************/
}
```

"strftime — Convert to Formatted Time" on page 557
"<monetary.h>" on page 771

strftime — Convert to Formatted Time

Format

```
#include <time.h>
size_t strftime(char *dest, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

Description

Language Level: ANSI, SAA, XPG4, POSIX

strftime converts the time and date specification in the *timeptr* structure into a character string. It then stores the null-terminated string in the array pointed to by *dest* according to the format string pointed to by *format*. *maxsize* specifies the maximum number of characters that can be copied into the array.

The format string is a multibyte character string containing:

Conversion specification characters, preceded by a % sign.
Ordinary multibyte characters, which are copied into the array unchanged.

If data has the form of a conversion specifier, but is not one of the accepted specifiers, the characters following the % are copied to the output.

The characters that are converted are determined by the LC_TIME category of the current locale and by the values in the time structure pointed to by *timeptr*. The time structure pointed to by *timeptr* is usually obtained by calling the gmtime or localtime function.

When objects to be copied overlap, the behavior is undefined.

strftime obtains time zone information from an internal structure. You can set the values in this structure by calling tzset which sets the structure according to the information in the TZ environment variable. If you do not set the values by calling this function, the defaults used are EST for time zone name, EDT for Daylight Savings time zone name, and 300 minutes for the difference between the standard time zone and the coordinated universal time.

The following table lists the strftime conversion specifiers:

Table 5 (Page 1 of 4). Conversion Specifiers Used by strftime

Specifier	Meaning
%a	Replace with abbreviated weekday name of locale.
%A	Replace with full weekday name of locale.
%b	Replace with abbreviated month name of locale.
%B	Replace with full month name of locale.

Table 5 (Page 2 of 4). Conversion Specifiers Used by strftime

Specifier	Meaning
%c	Replace with date and time of locale.
%C	Replace with locale's century number (year divided by 100 and truncated)
%d	Replace with day of the month (01-31).
%D	Insert date in mm/dd/yy form, regardless of locale.
%e	Insert month of the year as a decimal number (01–12). Under POSIX, it's a 2-character, right-justified, blank-filled field.
%E[cCxXyY]	If the alternate date/time format is not available, the %E descriptors are mapped to their unextended counterparts. For example, %EC is mapped to %C.
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's alternate representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's alternate representation.
%EY	Replace with the full alternative year representation.
%h	Replace with locale's abbreviated month name. This is the same as %b.
%H	Replace with hour (24-hour clock) as a decimal number (00-23).
%I	Replace with hour (12-hour clock) as a decimal number (01-12).
%j	Replace with day of the year (001-366).
%m	Replace with month (01-12).
%M	Replace with minute (00-59).
%n	Replace with a new line.
%O[deHImMSUwWy]	If the alternative date/time format is not available, the %O descriptors are mapped to their unextended counterparts. For example, %Od is mapped to %d.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero; otherwise fill with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.

Table 5 (Page 3 of 4). Conversion Specifiers Used by strftime

Specifier	Meaning
%OH	Replace with the hour (24-hour clock), using the locale's alternative numeric symbols.
%OI	Replace with the hour (12-hour clock), using the locale's alternative numeric symbols.
%Om	Replace with the month, using the locale's alternative numeric symbols.
%OM	Replace with the minutes, using the locale's alternative numeric symbols.
%OS	Replace with the seconds, using the locale's alternative numeric symbols.
%Ou	Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday, using the locale's alternative numeric symbols.
%OU	Replace with the week number of the year (00-53), where Sunday is the first day of the week, using the locale's alternative numeric symbols.
%OV	Replace with week number of the year (01-53), where Monday is the first day of the week, using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0), using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (01-53), where Monday is the first day of the week, using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation, using the locale's alternative numeric symbols.
%p	Replace with the locale's equivalent of AM or PM.
%r	Replace with a string equivalent to %I:%M:%S %p; or use t_fmt_ampm from LC_TIME, if present.
%R	Replace with time in 24 hour notation (%H:%M)
%S	Replace with second (00-61).
%t	Replace with a tab.
%T	Replace with a string equivalent to %H:%M:%S.
%u	Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday.
%U	Replace with week number of the year (00-53), where Sunday is the first day of the week.
%V	Replace with week number of the year (01-53), where Monday is the first day of the week.
%w	Replace with weekday (0-6), where Sunday is 0.

Table 5 (Page 4 of 4). Conversion Specifiers Used by strftime

Specifier	Meaning
%W	Replace with week number of the year (00-53), where Monday is the first day of the week.
%x	Replace with date representation of locale.
%X	Replace with time representation of locale.
%y	Replace with year without the century (00-99).
%Y	Replace with year including the century.
%Z	Replace with name of time zone, or no characters if time zone is not available.
%%	Replace with %.

For %Z, the tm_isdst flag in the tm structure passed to strftime specifies whether the time zone is standard or Daylight Savings time.

If data has the form of a directive, but is not one of the above, the characters following the % are copied to the output.

Return Value

If the total number of characters in the resulting string, including the terminating null character, does not exceed *maxsize*, strftime returns the number of characters (bytes) placed into *dest*, not including the terminating null character. Otherwise, strftime returns 0 and the content of the string is indeterminate.

This example gets the time and date from localtime, calls strftime to format it, and prints the resulting string.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char  dest[70];
   int  ch;
   time_t  temp;
    struct tm *timeptr;

    temp = time(NULL);
    timeptr = localtime(&temp);
    ch = strftime(dest, sizeof(dest)-1, "Today is %A,"" %b %d. \n Time: %I:%M %p"
        ,  timeptr);
    printf("%d characters placed in string to make: \n \n %s", ch, dest);
   return  0;

   /*****************************************************************************
        The output should be similar to:

        41 characters placed in string to make:
          Today is Monday, Sep 16.
          Time: 06:31 pm
   *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"setlocale — Set Locale" on page 499
"strptime — Convert to Formatted Date and Time" on page 576
"time — Determine Current Time" on page 625
"wcsftime — Convert to Formatted Date and Time" on page 719
"<time.h>" on page 779

stricmp — Compare Strings as Lowercase

Format

```
#include <string.h>
int stricmp(const char *string1, const char *string2);
```

Description

Language Level: Extension

stricmp compares *string1* and *string2* without sensitivity for case. All alphabetic characters in the arguments *string1* and *string2* are converted to lowercase before the comparison. stricmp operates on null-terminated strings.

Return Value

stricmp returns a value that indicates the following relationship between the two strings:

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* identical to *string2*
Greater than 0	*string1* greater than *string2*.

This example uses stricmp to compare two strings.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *str1 = "this is a string";
    char *str2 = "THIS IS A STRING";

    /* Compare two strings without regard to case                          */

    if (stricmp(str1, str2))
        printf("str1 is not the same as str2\n");
   else
        printf("str1 is the same as str2\n");
   return  0;

   /*************************************************************************
        The output should be:

        str1 is the same as str2
   *************************************************************************/
}
```

"strcmp — Compare Strings" on page 538
"strcmpi — Compare Strings Without Case Sensitivity" on page 540
"strcspn — Compare Strings for Substrings" on page 546
"strncmp — Compare Strings" on page 568
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"wcscmp — Compare Wide-Character Strings" on page 711
"wcsncmp — Compare Wide-Character Strings" on page 725

“<string.h>” on page 777

strlen — Determine String Length

Format

```
#include <string.h>
size_t strlen(const char *string);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strlen determines the length of *string* excluding the terminating null character.

Return Value

strlen returns the length of *string*.

This example determines the length of the string that is a constant within main.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
    char *String = "How long is this string?";

    printf("Length of string \"%s\" is %i.\n", String, strlen(String));
   return  0;

   /*****************************************************************************
        The output should be:

        Length of string "How long is this string?" is 24.
   *****************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"strrev — Reverse String" on page 583
"wcslen — Calculate Length of Wide-Character String" on page 722
"<string.h>" on page 777

strlwr — Convert Uppercase to Lowercase

Format

```
#include <string.h>
char *strlwr(char *string);
```

Description

Language Level: Extension

strlwr converts any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

Return Value

strlwr returns a pointer to the converted *string*. There is no error return.

This example makes a copy in all lowercase of the string "General Assembly", and then prints the copy.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *string = "General Assembly";
   char  *copy;

    copy = strlwr(strdup(string));
    printf("Expected result: general assembly\n");
    printf("strlwr returned: %s\n", copy);
   return  0;

  /*****************************************************************************
       The output should be:

       Expected result: general assembly
       strlwr returned: general assembly
  *****************************************************************************/
}
```

"strupr — Convert Lowercase to Uppercase" on page 604
"_toascii - _tolower - _toupper — Convert Character" on page 628
"tolower - toupper — Convert Character Case" on page 631
"towlower – towupper — Convert Wide Character Case" on page 632
"<string.h>" on page 777

strncat — Concatenate Strings

Format

#include <string.h>
char *strncat(char *_string1_, const char *_string2_, size_t _count_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strncat appends the first _count_ characters of _string2_ to _string1_ and ends the resulting string with a null character (\0). If _count_ is greater than the length of _string2_, the length of _string2_ is used in place of _count_.

The strncat function operates on null-terminated strings. The string argument to the function should contain a null character (\0) marking the end of the string.

Return Value

strncat returns a pointer to the joined string (_string1_).

This example demonstrates the difference between strcat and strncat. strcat appends the entire second string to the first, whereas strncat appends only the specified number of characters in the second string to the first.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char buffer1[SIZE] = "computer";
   char  *ptr;

    /* Call strcat with buffer1 and " program"                                  */

    ptr = strcat(buffer1, " program");
    printf("strcat : buffer1 = \"%s\"\n", buffer1);

    /* Reset buffer1 to contain just the string "computer" again                */

    memset(buffer1, '\0', sizeof(buffer1));
    ptr = strcpy(buffer1, "computer");

    /* Call strncat with buffer1 and " program"                                 */

    ptr = strncat(buffer1, " program", 3);
    printf("strncat: buffer1 = \"%s\"\n", buffer1);
   return  0;

   /*****************************************************************************
        The output should be:

        strcat : buffer1 = "computer program"
        strncat: buffer1 = "computer pr"
   *****************************************************************************/
}
```

"strcat — Concatenate Strings" on page 536
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"wcscat — Concatenate Wide-Character Strings" on page 708
"wcsncat — Concatenate Wide-Character Strings" on page 723
"<string.h>" on page 777

strncmp — Compare Strings

Format

```
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strncmp compares the first *count* characters of *string1* and *string2*. If *count* is greater than the length of *string1* or *string2*, characters that follow a null character are not compared.

Return Value

strncmp returns a value indicating the relationship between the substrings, as follows:

Value	Meaning
Less than 0	*substring1* less than *substring2*
0	*substring1* equivalent to *substring2*
Greater than 0	*substring1* greater than *substring2*

This example demonstrates the difference between strcmp and strncmp.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              10

int main(void)
{
    int  result;
     int index = 3;
     char buffer1[SIZE] = "abcdefg";
     char buffer2[SIZE] = "abcfg";
     void print_result(int, char *, char *);
```

```
    result = strcmp(buffer1, buffer2);
    printf("Comparison of each character\n");
   printf("    strcmp:  ");
    print_result(result, buffer1, buffer2);
    result = strncmp(buffer1, buffer2, index);
    printf("\nComparison of only the first %i characters\n", index);
   printf("    strncmp:  ");
    print_result(result, buffer1, buffer2);
   return  0;
   /*****************************************************************************
        The output should be:

        Comparison of each character
           strcmp: "abcdefg" is less than "abcfg"

        Comparison of only the first 3 characters
           strncmp: "abcdefg" is identical to "abcfg"
   *****************************************************************************/
}

void print_result(int res,char *p_buffer1,char *p_buffer2)
{
    if (0 == res)
        printf("\"%s\" is identical to \"%s\"\n", p_buffer1, p_buffer2);
   else
        if (res < 0)
            printf("\"%s\" is less than \"%s\"\n", p_buffer1, p_buffer2);
        else
            printf("\"%s\" is greater than \"%s\"\n", p_buffer1, p_buffer2);
}
```

"strcmp — Compare Strings" on page 538
"strcmpi — Compare Strings Without Case Sensitivity" on page 540
"strcoll — Compare Strings Using Collation Rules" on page 542
"strcspn — Compare Strings for Substrings" on page 546
"stricmp — Compare Strings as Lowercase" on page 562
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"wcscmp — Compare Wide-Character Strings" on page 711
"wcsncmp — Compare Wide-Character Strings" on page 725
"<string.h>" on page 777

strncpy — Copy Strings

Format

```
#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strncpy copies *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (\0) is *not* appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (\0) up to length *count*.

Return Value

strncpy returns a pointer to *string1*.

This example demonstrates the difference between strcpy and strncpy.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char source[SIZE] = "123456789";
    char source1[SIZE] = "123456789";
    char destination[SIZE] = "abcdefg";
    char destination1[SIZE] = "abcdefg";
   char  *return_string;
    int index = 5;

    /* This is how strcpy works                                          */

    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("After strcpy, destination becomes '%s'\n\n", destination);

    /* This is how strncpy works                                         */

    printf("destination1 is originally = '%s'\n", destination1);
    return_string = strncpy(destination1, source1, index);
    printf("After strncpy, destination1 becomes '%s'\n", destination1);
   return  0;
```

```
/*****************************************************************************
     The output should be:

     destination is originally = 'abcdefg'
     After strcpy, destination becomes '123456789'

     destination1 is originally = 'abcdefg'
     After strncpy, destination1 becomes '12345fg'
*****************************************************************************/
}
```

"strcpy — Copy Strings" on page 544
"strdup — Duplicate String" on page 549
"strnicmp — Compare Strings Without Case Sensitivity" on page 572
"<string.h>" on page 777

strnicmp — Compare Strings Without Case Sensitivity

Format

```
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);
```

Description

Language Level: Extension

strnicmp compares, at most, the first *n* characters of *string1* and *string2*. It operates on null-terminated strings.

strnicmp is case insensitive; the uppercase and lowercase forms of a letter are considered equivalent.

Return Value

strnicmp returns a value indicating the relationship between the substrings, as listed below:

Value	Meaning
Less than 0	*substring1* less than *substring2*
0	*substring1* equivalent to *substring2*
Greater than 0	*substring1* greater than *substring2*.

This example uses strnicmp to compare two strings.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *str1 = "THIS IS THE FIRST STRING";
    char *str2 = "This is the second string";
   int  numresult;

       /* Compare the first 11 characters of str1 and str2
            without regard to case                                          */

    numresult = strnicmp(str1, str2, 11);
    if (numresult < 0)
         printf("String 1 is less than string2.\n");
   else
         if (numresult > 0)
              printf("String 1 is greater than string2.\n");
        else
              printf("The two strings are equivalent.\n");
   return  0;

   /*************************************************************************
         The output should be:

         The two strings are equivalent.
   *************************************************************************/
}
```

"strcmp — Compare Strings" on page 538

"strcmpi — Compare Strings Without Case Sensitivity" on page 540
"stricmp — Compare Strings as Lowercase" on page 562
"strncmp — Compare Strings" on page 568
"wcscmp — Compare Wide-Character Strings" on page 711
"wcsncmp — Compare Wide-Character Strings" on page 725
"<string.h>" on page 777

strnset – strset — Set Characters in String

Format

```
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);
```

Description

Language Level: Extension

strnset sets, at most, the first *n* characters of *string* to *c* (converted to a char). If *n* is greater than the length of *string*, the length of *string* is used in place of *n*. strset sets all characters of *string*, except the ending null character (\0), to *c* (converted to a char).

For both functions, the *string* is a null-terminated string.

Return Value

Both strset and strnset return a pointer to the altered *string*. There is no error return value.

In this example, strnset sets not more than four characters of a string to the character 'x'. Then the strset function changes any non-null characters of the string to the character 'k'.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char str[] = "abcdefghi";

    printf("This is the string: %s\n", str);
    printf("This is the string after strnset: %s\n", strnset((char*)str, 'x', 4));
    printf("This is the string after strset: %s\n", strset((char*)str, 'k'));
   return  0;

  /*****************************************************************************
       The output should be:

       This is the string: abcdefghi
       This is the string after strnset: xxxxefghi
       This is the string after strset: kkkkkkkkk
  *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strpbrk — Find Characters in String" on page 575
"wcschr — Search for Wide Character" on page 709
"wcspbrk — Locate Wide Characters in String" on page 729
"<string.h>" on page 777

strpbrk — Find Characters in String

Format

```
#include <string.h>
char *strpbrk(const char *string1, const char *string2);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strpbrk locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

Return Value

strpbrk returns a pointer to the character. If *string1* and *string2* have no characters in common, a NULL pointer is returned.

This example returns a pointer to the first occurrence in the array *string* of either a or b.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *result,*string = "A Blue Danube";
    char *chars = "ab";

    result = strpbrk(string, chars);
    printf("The first occurrence of any of the characters \"%s\" in "
        "\"%s\" is \"%s\"\n", chars, string, result);
   return  0;

   /*****************************************************************************
        The output should be:

        The first occurrence of any of the characters "ab" in
        "A Blue Danube" is "anube"
   *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcspbrk — Locate Wide Characters in String" on page 729
"wcsrchr — Locate Wide Character in String" on page 731
"wcswcs — Locate Wide-Character Substring" on page 747
"<string.h>" on page 777

strptime — Convert to Formatted Date and Time

Format

#include <time.h>
char *strptime(const char **buf*, const char **fmt*, struct tm **tm*);

Description

Language Level: XPG4

strptime uses the format specified by *fmt* to convert the character string pointed to by *buf* to values that are stored in the structure pointed to by *tm*.

The **fmt* is composed of zero or more directives. Each directive is composed of one of the following:

One or more white-space characters (as specified by the isspace function)
An ordinary character (neither % nor a white-space character)
A conversion specifier.

Each conversion specifier consists of a % character followed by a conversion character that specifies the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifiers.

For a directive composed of white-space characters, strptime scans input up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

For a directive that is an ordinary character, strptime scans the next character from the buffer. If the scanned character differs from the one comprising the directive, the directive fails and the differing and subsequent characters remain unscanned.

For a series of directives composed of %n, %t, white-space characters, or any combination, strptime scans up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

For any other conversion specification, strptime scans characters until a character matching the next directive is scanned, or until no more characters can be scanned. It then compares these characters, excepting the one matching the next directive, to the locale values associated with the conversion specifier. If a match is found, strptime sets the appropriate tm structure members to values corresponding to the locale information. Case is ignored when items in *buf* are matched, such as month or weekday names. If no match is found, strptime fails and no more characters are scanned.

The following tables list the conversion specifiers for strptime.

Table 6 (Page 1 of 2). Conversion Specifiers Used by strptime

Specifier	Meaning
%a	Day of week, using locale's abbreviated or full weekday name.
%A	Day of week, using locale's abbreviated or full weekday name.
%b	Month, using locale's abbreviated or full month name.
%B	Month, using locale's abbreviated or full month name.
%c	Date and time, using locale's date and time.
%C	Century number (year divided by 100 and truncated to an integer)
%d	Day of the month (1-31; leading zeros permitted but not required).
%D	Date as %m/%d/%y.
%e	Day of the month (1-31; leading zeros permitted but not required).
%h	Month, using locale's abbreviated or full month name.
%H	Hour (0-23; leading zeros permitted but not required).
%I	Hour (0-12; leading zeros permitted but not required).
%j	Day number of the year (001-366).
%m	Month number (1-12; leading zeros permitted but not required).
%M	Minute (0-59; leading zeros permitted but not required).
%n	Newline character.
%p	Locale's equivalent of AM or PM.
%r	Time as %I:%M:%S a.m. or %I:%M:%S p.m.
%R	Time in 24 hour notation (%H%M)
%S	Seconds (0-61; leading zeros permitted but not required).
%t	Tab character.
%T	Time as %H:%M:%S.
%U	Week number of the year (0-53; where Sunday is the first day of the week; leading zeros permitted but not required).
%w	Weekday (0-6; where Sunday is 0; leading zeros permitted but not required).
%W	Week number of the year (0-53; where Monday is the first day of the week; leading zeros permitted but not required).
%x	Date, using locale's date format.
%X	Time, using locale's time format.

Table 6 (Page 2 of 2). Conversion Specifiers Used by strptime

Specifier	Meaning
%y	Year within century (0-99; leading zeros permitted but not required).
%Y	Year, including century.
%Z	Time zone name
%%	Replace with %.

Some directives can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

Table 7 (Page 1 of 2). Modified Directives Used by strptime

Specifier	Meaning
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's alternative representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's alternative representation.
%EY	Replace with the full alternative year representation.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero; otherwise, fill with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock), using the locale's alternative numeric symbols.
%OI	Replace with the hour (12-hour clock), using the locale's alternative numeric symbols.
%Om	Replace with the month, using the locale's alternative numeric symbols.
%OM	Replace with the minutes, using the locale's alternative numeric symbols.
%OS	Replace with the seconds, using the locale's alternative numeric symbols.

Table 7 (Page 2 of 2). Modified Directives Used by strptime

Specifier	Meaning
%OU	Replace with the week number of the year (Sunday as the first day of the week, rules corresponding to %U), using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0), using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week), using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation, using the locale's alternative numeric symbols.

Return Value

If successful, strptime returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

This example uses strptime to convert a string to the structure xmas, then prints the contents of the structure.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    struct tm xmas;

    if (NULL == strptime("12/25/94 12:00:01", "%D %T", &xmas)) {
        printf("strptime()  failed.\n");
        exit(EXIT_FAILURE);
    }
    printf("tm_sec    = %3d\n", xmas.tm_sec );
    printf("tm_min    = %3d\n", xmas.tm_min );
    printf("tm_hour = %3d\n", xmas.tm_hour);
    printf("tm_mday = %3d\n", xmas.tm_mday);
    printf("tm_mon    = %3d\n", xmas.tm_mon );
    printf("tm_year = %3d\n", xmas.tm_year);
   return  0;

   /*****************************************************************************
        The output should be similar to :

       tm_sec   =    1
       tm_min   =    0
       tm_hour  =   11
       tm_mday  =   25
        tm_mon  =   12
       tm_year  =   94
   *****************************************************************************/
}
```

strptime

"strftime — Convert to Formatted Time" on page 557
"wcsftime — Convert to Formatted Date and Time" on page 719
"<time.h>" on page 779

strrchr — Find Last Occurrence of Character in String

Format

```
#include <string.h>
char *strrchr(const char *string, int c);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

strrchr finds the last occurrence of *c* (converted to a character) in *string*. The ending null character is considered part of the *string*.

Return Value

strrchr returns a pointer to the last occurrence of *c* in *string*. If the given character is not found, a NULL pointer is returned.

This example compares the use of strchr and strrchr. It searches the string for the first and last occurrence of p in the string.

```
#include <stdio.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    char buf[SIZE] = "computer program";
   char  *ptr;
    int ch = 'p';

    /* This illustrates strchr                                                */

    ptr = strchr(buf, ch);
    printf("The first occurrence of %c in '%s' is '%s'\n", ch, buf, ptr);

    /* This illustrates strrchr                                               */

    ptr = strrchr(buf, ch);
    printf("The last occurrence of %c in '%s' is '%s'\n", ch, buf, ptr);
   return  0;

   /*****************************************************************************
        The output should be:

        The first occurrence of p in 'computer program' is 'puter program'
        The last occurrence of p in 'computer program' is 'program'
   ******************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcspbrk — Locate Wide Characters in String" on page 729

strrchr

"wcsrchr — Locate Wide Character in String" on page 731
"wcswcs — Locate Wide-Character Substring" on page 747
"<string.h>" on page 777

strrev — Reverse String

Format

```
#include <string.h>
char *strrev(char *string);
```

Description

Language Level: Extension

strrev reverses the order of the characters in the given *string*. The ending null character (\0) remains in place.

Return Value

strrev returns a pointer to the altered *string*. There is no error return value.

This example determines whether a string is a *palindrome*. A palindrome is a string that reads the same forward and backward.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int palindrome(char *string)
{
    char  *string2;

    /* Duplicate string for comparison                                    */

    if (NULL == (string2 = strdup(string))) {
        printf("Storage could not be reserved for string\n");
        exit(EXIT_FAILURE);
    }

    /* If result equals 0, the string is a palindrome                     */

    return (strcmp(string, strrev(string2)));
}
```

strrev

```
int main(void)
{
    char  string[81];

     printf("Please enter a string.\n");
    scanf("%80s",  string);
    if  (palindrome(string))
          printf("The string is not a palindrome.\n");
    else
          printf("The string is a palindrome.\n");
    return  0;

    /**************************************************************************
          Sample output from program:

          Please enter a string.
          level
          The string is a palindrome.
          ... or ...
          Please enter a string.
          levels
          The string is not a palindrome.
    **************************************************************************/
}
```

"strcat — Concatenate Strings" on page 536
"strcmp — Compare Strings" on page 538
"strcpy — Copy Strings" on page 544
"strdup — Duplicate String" on page 549
"strnset – strset — Set Characters in String" on page 574
"<string.h>" on page 777

strspn — Search Strings

Format

#include <string.h>
size_t strspn(const char *_string1_, const char *_string2_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strspn finds the first occurrence of a character in _string1_ that is not contained in the set of characters specified by _string2_. The null character (\0) that ends _string2_ is not considered in the matching process.

Return Value

strspn returns the index of the first character found. This value is equal to the length of the initial substring of _string1_ that consists entirely of characters from _string2_. If _string1_ begins with a character not in _string2_, strspn returns 0. If all the characters in _string1_ are found in _string2_, the length of _string1_ is returned.

This example finds the first occurrence in the array _string_ of a character that is not an a, b, or c. Because the string in this example is cabbage, strspn returns 5, the length of the segment of cabbage before a character that is not an a, b, or c.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string = "cabbage";
    char *source = "abc";
   int  index;

    index = strspn(string, "abc");
    printf("The first %d characters of \"%s\" are found in \"%s\"\n", index,
        string,  source);
   return  0;

   /*****************************************************************************
        The output should be:

        The first 5 characters of "cabbage" are found in "abc"
   *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcspbrk — Locate Wide Characters in String" on page 729
"wcsrchr — Locate Wide Character in String" on page 731
"wcsspn — Search Wide-Character Strings" on page 735

"wcswcs — Locate Wide-Character Substring" on page 747
"<string.h>" on page 777

strstr — Locate Substring

Format

#include <string.h>
char *strstr(const char *_string1_, const char *_string2_);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strstr finds the first occurrence of _string2_ in _string1_. The function ignores the null character (\0) that ends _string2_ in the matching process.

Return Value

strstr returns a pointer to the beginning of the first occurrence of _string2_ in _string1_. If _string2_ does not appear in _string1_, strstr returns NULL. If _string2_ points to a string with zero length, strstr returns _string1_.

This example locates the string haystack in the string "needle in a haystack".

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1 = "needle in a haystack";
    char *string2 = "haystack";
   char  *result;

    result = strstr(string1, string2);

    /* Result = a pointer to "haystack"                                    */

   printf("%s\n",  result);
   return  0;

   /*****************************************************************************
       The output should be:

      haystack
   *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcspbrk — Locate Wide Characters in String" on page 729
"wcsrchr — Locate Wide Character in String" on page 731
"wcsspn — Search Wide-Character Strings" on page 735
"wcswcs — Locate Wide-Character Substring" on page 747

strstr

"<string.h>" on page 777

_strtime — Copy Time

Format

```
#include <time.h>
char *_strtime(char *time);
```

Description

Language Level: Extension

_strtime copies the current time into the buffer that *time* points to. The format is:

hh:mm:ss

where

hh represents the hour in 24-hour notation,
mm represents the minutes past the hour,
ss represents the number of seconds.

For example, the string 18:23:44 represents 23 minutes and 44 seconds past 6 p.m.

The buffer must be at least 9 bytes.

Return Value

_strtime returns a pointer to the buffer. There is no error return.

This example prints the current time:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char  buffer[9];

    printf("The current time is %s \n", _strtime(buffer));
   return  0;

   /*************************************************************************
       The output should be similar to:

       The current time is 16:47:22
   *************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"time — Determine Current Time" on page 625
"tzset — Assign Values to Locale Information" on page 634
"<time.h>" on page 779

strtod — Convert Character String to Double

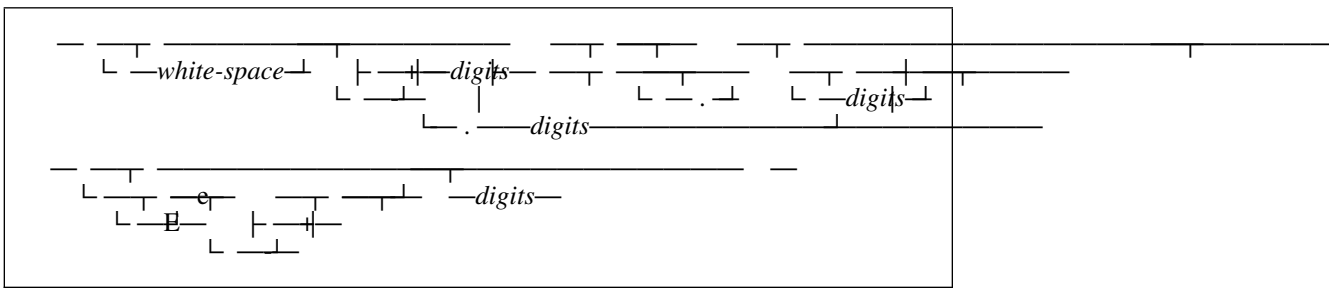
Format

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Description

Language Level: ANSI, SAA, XPG4

strtod converts a character string to a double-precision value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type double. When it reads a character that it does not recognize as part of a number, strtod stops reading the string at that character, and sets endptr to point to the remainder of nptr. The character at which strtod stops reading the string may be the null character at the end of that string. The strtod function expects *nptr* to point to a string with the following form:



Note: The character used for the decimal point (shown as . in the above diagram) depends on the LC_NUMERIC category of the current locale.

The first character that does not fit this form stops the scan.

Return Value

strtod returns the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, it returns -HUGE_VAL or +HUGE_VAL; for an underflow, it returns 0.

In both cases, errno is set to ERANGE, depending on the base of the value. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

strtod does not fail if a character other than a digit follows an E or e read in as an exponent. For example, 100elf will be converted to the floating-point value 100.0.

This example converts the strings to a double value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *string,*stopstring;
   double  x;

    string = "3.1415926This stopped it";
    x = strtod(string, &stopstring);
    printf("string = %s\n", string);
   printf("    strtod  =  %f\n",  x);
   printf("    Stopped  scan  at  %s\n\n",  stopstring);
    string = "100ergs";
    x = strtod(string, &stopstring);
    printf("string = \"%s\"\n", string);
   printf("    strtod  =  %f\n",  x);
   printf("    Stopped  scan  at  \"%s\"\n\n",  stopstring);
   return  0;

   /*****************************************************************************
        The output should be:

        string = 3.1415926This stopped it
            strtod = 3.141593
            Stopped scan at This stopped it

        string = "100ergs"
            strtod = 100.000000
            Stopped scan at "ergs"
   *****************************************************************************/
}
```

“atof — Convert Character String to Float” on page 56
“atoi — Convert Character String to Integer” on page 58
“atol — Convert Character String to Long Integer” on page 60
“_atold — Convert Character String to Long Double” on page 64
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“strtoul — Convert String Segment to Unsigned Integer” on page 600
“wcstod — Convert Wide-Character String to Double” on page 737
“wcstol — Convert Wide-Character to Long Integer” on page 741
“wcstoul — Convert Wide-Character String to Unsigned Long” on page 745
“<stdlib.h>” on page 775

strtok — Tokenize String

Format

#include <string.h>
char *strtok(char **string1*, const char **string2*);

Description

Language Level: ANSI, SAA, POSIX, XPG4

strtok reads *string1* as a series of zero or more tokens, and *string2* as the set of characters serving as delimiters of the tokens in *string1*. The tokens in *string1* can be separated by one or more of the delimiters from *string2*. The tokens in *string1* can be located by a series of calls to strtok.

In the first call to strtok for a given *string1*, strtok searches for the first token in *string1*, skipping over leading delimiters. A pointer to the first token is returned.

To read the next token from *string1*, call strtok with a NULL *string1* argument. A NULL *string1* argument causes strtok to search for the next token in the previous token string. Each delimiter is replaced by a null character. The set of delimiters can vary from call to call, so *string2* can take any value.

Return Value

The first time strtok is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, strtok returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-terminated.

Using a loop, this example gathers tokens, separated by commas, from a string until no tokens are left. After processing the tokens (not shown), the example returns the pointers to the tokens a string, of, tokens and a blank. The next call to strtok returns NULL, and the loop ends.

```
#pragma strings(writable)
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *token,string[] = "a string, of, ,tokens\0,after null terminator";

     /* the string pointed to by string is broken up into the tokens
          "a string", " of", " ", and "tokens" ; the null terminator (\0)
          is encountered and execution stops after the token "tokens"                    */

    token = strtok((char*)string, ",");
   do  {
         printf("token: %s\n", token);
    }    while (token = strtok(NULL, ","));
   return  0;

   /*************************************************************************
         The output should be:

         token: a string
        token:    of
        token:
        token:  tokens
   *************************************************************************/
}
```

“strcat — Concatenate Strings” on page 536
“strchr — Search for Character” on page 537
“strcmp — Compare Strings” on page 538
“strcpy — Copy Strings” on page 544
“strcspn — Compare Strings for Substrings” on page 546
“strspn — Search Strings” on page 585
“wcstok — Tokenize Wide-Character String” on page 739
“<string.h>” on page 777

strtol — Convert Character String to Long Integer

Format

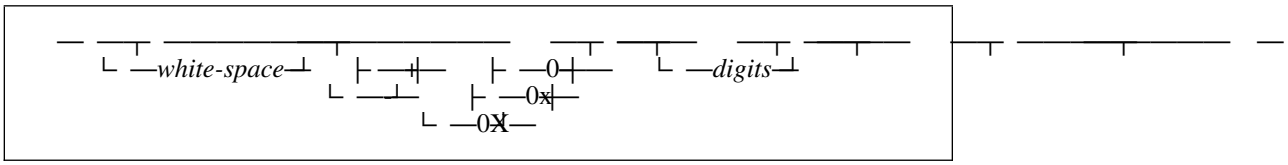
```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description

Language Level: ANSI, SAA, XPG4

strtol converts a character string to a long-integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type long int. When it reads a character that it does not recognize as part of a number, strtol stops reading the string at that character, and sets endptr to point to the remainder of nptr. The character at which strtol stops reading the string may be the null character at the end of that string. The ending character can also be the first numeric character greater than or equal to the *base*.

When you use the strtol function, *nptr* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Return Value

strtol returns the value represented in the string, except when the representation causes an overflow. For an overflow, it returns LONG_MAX or LONG_MIN, according to the sign of the value and errno is set to ERANGE. If *base* is not a valid number, strtol sets errno to EDOM.

If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

This example converts the strings to a long value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *string,*stopstring;
   long  l;
   int  bs;

    string = "10110134932";
    printf("string = %s\n", string);
    for (bs = 2; bs <= 8; bs *= 2) {
        l = strtol(string, &stopstring, bs);
        printf("      strtol = %ld (base %d)\n", l, bs);
        printf("      Stopped scan at %s\n\n", stopstring);
   }
   return  0;

   /*****************************************************************************
        The output should be:

        string = 10110134932
            strtol = 45 (base 2)
            Stopped scan at 34932

        strtol = 4423 (base 4)
        Stopped scan at 4932

        strtol = 2134108 (base 8)
        Stopped scan at 932
   *****************************************************************************/
}
```

"atof — Convert Character String to Float" on page 56
"atoi — Convert Character String to Integer" on page 58
"atol — Convert Character String to Long Integer" on page 60
"_atold — Convert Character String to Long Double" on page 64
"_ltoa — Convert Long Integer to String" on page 367
"strtod — Convert Character String to Double" on page 590
"strtold — Convert String to Long Double" on page 596
"strtoll — Convert Character String to Long Long Integer" on page 598
"strtoul — Convert String Segment to Unsigned Integer" on page 600
"wcstod — Convert Wide-Character String to Double" on page 737
"wcstol — Convert Wide-Character to Long Integer" on page 741
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"<stdlib.h>" on page 775

strtold — Convert String to Long Double

Format

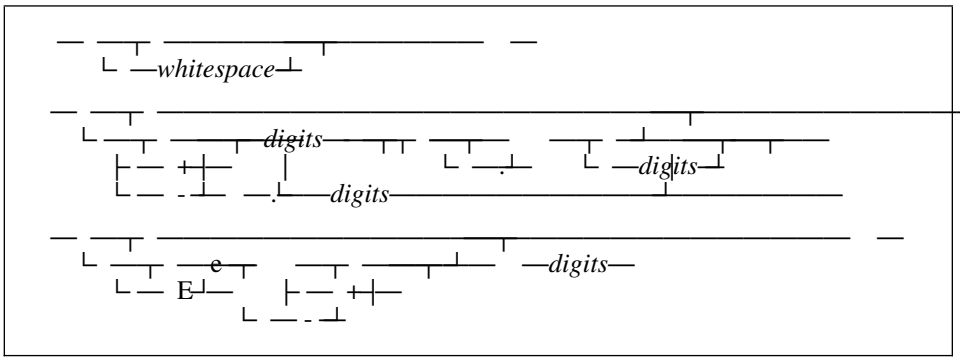
#include <stdlib.h>
long double strtold(const char **nptr*, char ***endptr*);

Description

Language Level: Extension

strtold converts a character string to a long double value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type long double. When it reads a character that it does not recognize as part of a number, strtold stops reading the string at that character, and sets endptr to point to the remainder of nptr. The character at which strtod stops reading the string may be the null character at the end of that string.

The string pointed to by *nptr* must have the following format:



The *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one digit must follow the decimal point. An exponent expressed as a decimal integer can follow the digits. The exponent can be signed.

The value of *nptr* can also be one of the strings infinity, inf, or nan. These strings are case insensitive, and can be preceded by a unary minus (-). They are converted to infinity and NaN values. See “Infinity and NaN Support” on page 27 for more information about using infinity and NaN values.

If the string pointed to by *nptr* does not have the expected form, no conversion is performed and *endptr* points to the value of *nptr*.

Return Value If successful, strtold returns the value of the long double number. If it fails, strtold returns 0. For an underflow or overflow, it returns the following:

Condition	Return Value
Underflow	0 with errno set to ERANGE
Positive overflow	+_LHUGE_VAL
Negative overflow	-_LHUGE_VAL.

This example uses strtold to convert two strings, " -001234.5678e10end of string" and "NaNthis cannot be converted" to their corresponding long double values. It also prints out the part of the string that cannot be converted.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *nptr;
   char  *endptr;

    nptr = "    -001234.5678e10end of string";
    printf("strtold = %.10Le\n", strtold(nptr, &endptr));
    printf("end pointer at = %s\n\n", endptr);
    nptr = "NaNthis cannot be converted";
    printf("strtold = %.10Le\n", strtold(nptr, &endptr));
    printf("end pointer at = %s\n\n", endptr);
   return  0;

   /*****************************************************************************
        The output should be:

        strtold = -1.2345678000e+13
        end pointer at = end of string

        strtold = nan
        end pointer at = this cannot be converted
   *****************************************************************************/
}
```

"atof — Convert Character String to Float" on page 56
"atoi — Convert Character String to Integer" on page 58
"atol — Convert Character String to Long Integer" on page 60
"_atold — Convert Character String to Long Double" on page 64
"strtod — Convert Character String to Double" on page 590
"strtol — Convert Character String to Long Integer" on page 594
"strtoul — Convert String Segment to Unsigned Integer" on page 600
"wcstod — Convert Wide-Character String to Double" on page 737
"wcstol — Convert Wide-Character to Long Integer" on page 741
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"Infinity and NaN Support" on page 27
"<stdlib.h>" on page 775

strtoll — Convert Character String to Long Long Integer

Format

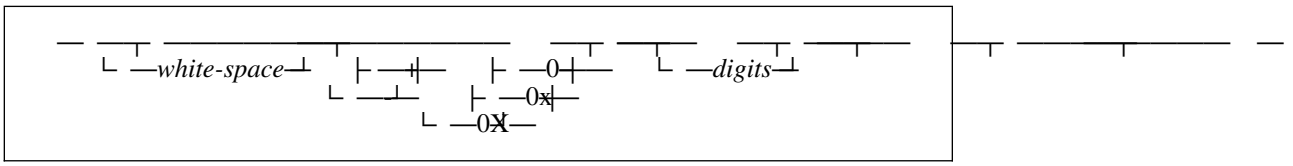
#include <stdlib.h>
long long strtoll(const char **nptr*, char ***endptr*, int *base*);

Description

Language Level: Extension

strtoll converts a character string to a long long value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of type long long int. This function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the null character (\0) at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base*.

When you use the strtoll function, *nptr* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Return Value

strtoll returns the value represented in the string, except when the representation causes an overflow. For an overflow, it returns LONGLONG_MAX or LONGLONG_MIN, according to the sign of the value and errno is set to ERANGE. If *base* is not a valid number, strtoll sets errno to EDOM.

errno is set to ERANGE for the exceptional cases, depending on the base of the value. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

This example converts the strings to a long long value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char  *string,*stopstring;
    long long ll;
   int  bs;

    string = "10110134932";
    printf("string = %s\n\n", string);
    for (bs = 2; bs <= 8; bs *= 2) {
        ll = strtoll(string, &stopstring, bs);
        printf("      strtoll = %lld (base %d)\n", ll, bs);
        printf("      Scan stopped at %s\n\n", stopstring);
   }
   return  0;

   /*****************************************************************************
        The output should be:

        string = 10110134932

            strtoll = 45 (base 2)
            Scan stopped at 34932

            strtoll = 4423 (base 4)
            Scan stopped at 4932

            strtoll = 2134108 (base 8)
            Scan stopped at 932
   *****************************************************************************/
}
```

"atof — Convert Character String to Float" on page 56
"atoi — Convert Character String to Integer" on page 58
"atol — Convert Character String to Long Integer" on page 60
"_atold — Convert Character String to Long Double" on page 64
"_ltoa — Convert Long Integer to String" on page 367
"strtod — Convert Character String to Double" on page 590
"strtol — Convert Character String to Long Integer" on page 594
"strtold — Convert String to Long Double" on page 596
"strtoul — Convert String Segment to Unsigned Integer" on page 600
"strtoull — Convert String Segment to Unsigned Long Long Integer" on page 602
"wcstod — Convert Wide-Character String to Double" on page 737
"wcstol — Convert Wide-Character to Long Integer" on page 741
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"<stdlib.h>" on page 775

strtoul — Convert String Segment to Unsigned Integer

Format

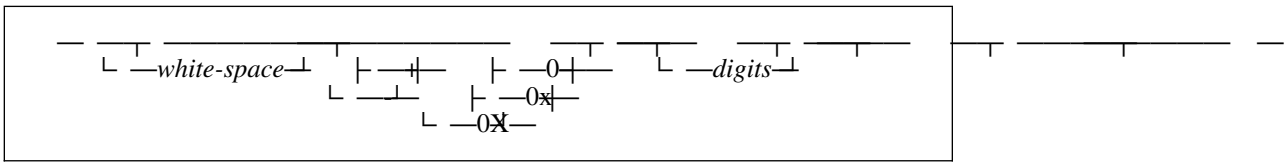
#include <stdlib.h>
unsigned long int strtoul(const char **string1*, char ***string2*, int *base*);

Description

Language Level: ANSI, SAA, XPG4

strtoul converts a character string to an unsigned long integer value. The input *string1* is a sequence of characters that can be interpreted as a numerical value of the type unsigned long int. strtoul stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. strtoul sets *string2* to point to the resulting output string if a conversion is performed, and provided that *string2* is not a NULL pointer.

When you use strtoul, *string1* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Return Value

strtoul returns the value represented in the string, or 0 if no conversion could be performed. For an overflow, strtoul returns ULONG_MAX and sets errno to ERANGE. If *base* is not a valid number, strtoul sets errno to EDOM.

This example converts the string to an unsigned long value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdio.h>
#include <stdlib.h>

#define   BASE              2

int main(void)
{
   char  *string,*stopstring;
    unsigned long ul;

    string = "1000e13 e";
    printf("string = %s\n", string);
    ul = strtoul(string, &stopstring, BASE);
   printf("    strtoul = %ld (base %d)\n", ul, BASE);
   printf("    Stopped scan at %s\n\n", stopstring);
   return  0;

   /*************************************************************************
        The output should be:

        string = 1000e13 e
            strtoul = 8 (base 2)
            Stopped scan at e13 e
   *************************************************************************/
}
```

"atof — Convert Character String to Float" on page 56
"atoi — Convert Character String to Integer" on page 58
"atol — Convert Character String to Long Integer" on page 60
"_atold — Convert Character String to Long Double" on page 64
"strtod — Convert Character String to Double" on page 590
"strtol — Convert Character String to Long Integer" on page 594
"strtold — Convert String to Long Double" on page 596
"strtoll — Convert Character String to Long Long Integer" on page 598
"strtoull — Convert String Segment to Unsigned Long Long Integer" on page 602
"_ultoa — Convert Unsigned Long Integer to String" on page 675
"wcstod — Convert Wide-Character String to Double" on page 737
"wcstol — Convert Wide-Character to Long Integer" on page 741
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"<stdlib.h>" on page 775

strtoull — Convert String Segment to Unsigned Long Long Integer

Format

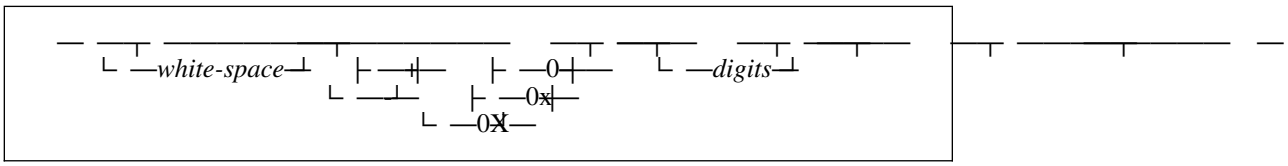
#include <stdlib.h>
unsigned long long int strtoull(const char **string1*, char ***string2*, int *base*);

Description

Language Level: Extension

strtoull converts a character string to an unsigned long long integer value. The input *string1* is a sequence of characters that can be interpreted as a numerical value of the type unsigned long long int. strtoull stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. strtoull sets *string2* to point to the resulting output string if a conversion is performed, and provided that *string2* is not a NULL pointer.

When you use strtoull, *string1* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Return Value

strtoull returns the value represented in the string, or 0 if no conversion could be performed. For an overflow, strtoull returns ULONGLONG_MAX and sets errno to ERANGE. If *base* is not a valid number, strtoull sets errno to EDOM.

This example converts the string to an unsigned long long value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdio.h>
#include <stdlib.h>

#define   BASE              2

int main(void)
{
   char  *string,*stopstring;
    unsigned long long ull;

    string = "1000e13 e";
    printf("string = %s\n", string);
    ull = strtoull(string, &stopstring, BASE);
   printf("    strtoull = %lld (base %d)\n", ull, BASE);
   printf("    Scan stopped at %s\n\n", stopstring);
   return  0;

   /*****************************************************************************
        The output should be:

        string = 1000e13 e
            strtoul = 8 (base 2)
            Scan stopped at e13 e
   *****************************************************************************/
}
```

“atof — Convert Character String to Float” on page 56
“atoi — Convert Character String to Integer” on page 58
“atol — Convert Character String to Long Integer” on page 60
“_atold — Convert Character String to Long Double” on page 64
“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtoll — Convert Character String to Long Long Integer” on page 598
“strtold — Convert String to Long Double” on page 596
“_ultoa — Convert Unsigned Long Integer to String” on page 675
“wcstod — Convert Wide-Character String to Double” on page 737
“wcstol — Convert Wide-Character to Long Integer” on page 741
“wcstoul — Convert Wide-Character String to Unsigned Long” on page 745
“<stdlib.h>” on page 775

strupr — Convert Lowercase to Uppercase

Format

```
#include <string.h>
char *strupr(char *string);
```

Description

Language Level: Extension

strupr converts any lowercase letters in *string* to uppercase. Other characters are not affected.

Return Value

strupr returns a pointer to the converted *string*. There is no error return.

This example makes a copy in all-uppercase of the string "Win32", and then prints the copy.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *string = "Hello World!";
   char  *copy;

    copy = strupr(strdup(string));
    printf("This is a copy of the string with all letters capitalized: %s\n",
               copy);
   return  0;

   /*****************************************************************************
       The output should be:

       This is a copy of the string with all letters capitalized: HELLO WORLD!
   *****************************************************************************/
}
```

"strlwr — Convert Uppercase to Lowercase" on page 565
"_toascii - _tolower - _toupper — Convert Character" on page 628
"tolower - toupper — Convert Character Case" on page 631
"towlower – towupper — Convert Wide Character Case" on page 632
"<string.h>" on page 777

strxfrm — Transform String

Format

```
#include <string.h>
size_t strxfrm(char *str1, const char *str2, size_t n);
```

Description

Language Level: ANSI, SAA, XPG4

strxfrm transforms the string pointed to by *str2* and places the resulting string into the array pointed to by *str1*. The transformation is determined by the program's locale. The transformed string may not be displayable or printable, but can be used with the strcmp or strncmp functions.

The result of applying strcoll to two separate strings before their transformation is equal to the result of applying strcmp, or strncmp, to the same two strings after their transformation.

No more than *n* bytes are placed into the area pointed to by *str1*, including the terminating null byte. If *n* is 0, *str1* can be a null pointer.

Return Value

strxfrm returns the length of the transformed string (excluding the null byte). When *n* is 0 and *str1* is a null pointer, the length returned is one less than the number of bytes required to contain the transformed string. If an error occurs, strxfrm function returns (size_t)−1 and sets **errno** to indicate the error.

Notes:

1. The string returned by strxfrm contains the weights for each order of the characters within the string. As a result, the string returned may be longer than the input string; it does not contain printable characters.
2. strxfrm calls malloc when the LC_COLLATE category specifies *backward* on the *order_start* keyword, the *substitute* keyword is specified, or the locale has one-to-many mapping. If malloc fails, strxfrm also fails.
3. If the locale supports double-byte characters (MB_CUR_MAX specified as 2), strxfrm validates the multibyte characters. (Previously it did not validate the string.) strxfrm will fail if the string contains invalid multibyte characters.
4. If MB_CUR_MAX is defined as 2, and no collation is defined for DBCS chars in the current locale, the DBCS characters will collate after the single-byte characters.

This example uses strxfrm to transform two different strings that have the same collating weight. It then calls strcmp to compare the new strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <string.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "da_dk.ibm-865"        /* OS/2 name              */
    char *string1 = "str\xA0ng1a";
    char *string2 = "strang1\x83";
#else
    #define LOCNAME "da_dk.ibm-1252"       /* Windows name           */
    char *string1 = "str\xE0ng1a";
    char *string2 = "strang1\xE2";
#endif

int main(void)
{
    char *newstring1, *newstring2;
    size_t length1, length2, pw1, pw2;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
    }
    length1 = strxfrm(NULL, string1, 0);
    pw1 = strlen(string1);
    length2 = strxfrm(NULL, string2, 0);
    pw2 = strlen(string2);
    if (NULL == (newstring1 =(char*) calloc(length1 + 1, 1)) ||
          NULL == (newstring2 =(char*) calloc(length2 + 1, 1))) {
        printf("insufficient  memory\n");
        exit(1);
    }
```

```
    /* Get primary weight of each string                                    */
    if ((strxfrm(newstring1, string1, pw1 + 1) != length1) ||
          (strxfrm(newstring2, string2, pw2 + 1) != length2)) {
        printf("error in string processing\n");
        exit(1);
    }
    if (0 != strcmp(newstring1, newstring2))
        printf("wrong  results\n");
    else
        printf("correct  results\n");
    return  0;

    /*************************************************************************
        The output should be similar to :

        correct  results
    *************************************************************************/
}
```

"localeconv — Retrieve Information from the Environment" on page 352
"setlocale — Set Locale" on page 499
"strcmp — Compare Strings" on page 538
"strcoll — Compare Strings Using Collation Rules" on page 542
"strncmp — Compare Strings" on page 568
"wcsxfrm — Transform Wide-Character String" on page 749
"<string.h>" on page 777

swab — Swap Adjacent Bytes

Format

#include <stdlib.h>
void swab(char *_source_, char *_destination_, int _n_);

Description

Language Level: Extension

swab copies _n_ bytes from _source_, swaps each pair of adjacent bytes, and stores the result at _destination_. The integer _n_ should be an even number to allow for swapping. If _n_ is an odd number, a null character (\0) is added after the last byte.

swab is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Note: In earlier releases of VisualAge C++, swab began with an underscore (_swab). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _swab to swab for you.

Return Value

There is no return value.

This example copies _n_ bytes from one location to another, swapping each pair of adjacent bytes. In the output _x_ is replaced with null character.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char from[21] = "hTsii    s atsirgn..x ";
   char  to[21];

    swab(from, to, 19);    /* swap bytes */
   printf("%s\n",  to);

   return  0;

   /*****************************************************************************
        The output should be:

        This is a string..
   *****************************************************************************/
}
```

"fgetc — Read a Character" on page 200
"fputc — Write Character" on page 227
"<stdlib.h>" on page 775

swprintf — Format and Write Wide Characters to Buffer

Format

```
#include <wchar.h>
int swprintf(wchar_t *wcsbuffer, size_t n,
                    const wchar_t *format, argument-list);
```

Description

Language Level: ANSI 93

swprintf formats and stores a series of wide characters and values into the wide-character buffer *wcsbuffer*. swprintf is equivalent to sprintf, except that it operates on wide characters.

The value *n* specifies the maximum number of wide characters to be written, including the terminating null character.

swprintf converts each entry in the *argument-list* according to the corresponding wide-character format specifier in *format*. The *format* has the same form and function as the format string for printf, with the following exceptions:

%c (without an l prefix) converts an integer argument to **wchar_t**, as if by calling mbtowc.
%lc converts a **wint_t** to **wchar_t**.
%s (without an l prefix) converts an array of multibyte characters to an array of **wchar_t**, as if by calling mbrtowc. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.
%ls writes an array of **wchar_t**. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.

If you specify a null string for the %s or %ls format specifier, swprintf prints (null).

For a complete description of format specifiers, see “printf — Print Formatted Characters” on page 429.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

In extended mode, swprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See “Infinity and NaN Support” on page 27 for more information on infinity and NaN values.

Return Value swprintf returns the number of bytes written in the array, not counting the terminating null wide character.

This example uses swprintf to format and print several values to buffer.

```
#include <wchar.h>
#include <stdio.h>

#define BUF_SIZE 100

int main(void)
{
    wchar_t  wcsbuf[BUF_SIZE];
     wchar_t wstring[] = L"ABCDE";
    int         num;

     num = swprintf(wcsbuf, BUF_SIZE, L"%s", "xyz");
     num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%ls", wstring);
     num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%i", 100);
     printf("The array wcsbuf contains: \"%ls\"\n", wcsbuf);
    return  0;

   /*************************************************************************
        The output should be similar to :

        The array wcsbuf contains: "xyzABCDE100"
   *************************************************************************/
}
```

"printf — Print Formatted Characters" on page 429
"sprintf — Print Formatted Data to Buffer" on page 525
"swscanf — Read Wide Characters from Buffer" on page 611
"vswprintf — Format and Write Wide Characters to Buffer" on page 704
"<wchar.h>" on page 780

swscanf — Read Wide Characters from Buffer

Format

```
#include <wchar.h>
int swscanf(const wchar_t *wcsbuffer, const wchar_t *format,
            argument-list);
```

Description

Language Level: ANSI 93

swscanf reads wide data from *wcsbuffer* into the locations given by *argument-list*. swscanf is equivalent to sscanf, except that it operates on wide characters.

Each argument in the *argument-list* must point to a variable with a type that corresponds to a type specifier in *format*. The *format* has the same form and function as the format string for scanf, with the following exceptions:

%c (with no l prefix) converts one or more **wchar_t** characters (depending on precision) to multibyte characters, as if by calling wcrtomb.
%lc converts one or more **wchar_t** characters (depending on precision) to an array of **wchar_t**.
%s (with no l prefix) converts a sequence of non-white-space **wchar_t** characters to multibyte characters, as if by calling wcrtomb. The array includes the terminating null character.
%ls copies an array of **wchar_t**, including the terminating null wide character, to an array of **wchar_t**.

For a complete description of format specifiers, see “scanf — Read Data” on page 486.

When swscanf reaches the end of the wide character string, it stops parsing and returns a value as indicated below (this behavior is the same as sscanf). If copying takes place between objects that overlap, the behavior is undefined.

Return Value

swscanf returns the number of input items that were successfully converted and assigned. The value does not include fields that were read but not assigned. If an input failure (such as the end of the string) is encountered before any conversion, swscanf returns EOF.

swscanf

This example uses swscanf to scan in wide characters, and then prints them.

```
#include <wchar.h>
#include <stdio.h>

int main(void)
{
    wchar_t *tokenstring = L"xyz âAâBâCâDâE a 1234";
   char       string[20];
   wchar_t  wstring[20];
   wchar_t  wc;
   int         i;
   int         num;

    num = swscanf(tokenstring, L"%s %ls %lc %d", string, wstring, &wc, &i);
   printf("string    =  %s\n",    string  );
    printf("wstring   = %ls\n", wstring );
    printf("wc             = %lc\n", wc );
   printf("i              =  %d\n",    i  );
    printf("total number of fields scanned: %i\n", num);
   return  0;

   /*****************************************************************************
        The output should be similar to :

       string     =  xyz
       wstring    =  âAâBâCâDâE a
       wc           =  1
       i             =  234
        total number of fields scanned: 4
   *****************************************************************************/
}
```

"scanf — Read Data" on page 486
"sscanf — Read Data" on page 530
"swprintf — Format and Write Wide Characters to Buffer" on page 609
"<wchar.h>" on page 780

system — Invoke the Command Processor

Format

```
#include <stdlib.h>
int system(char *string);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

system passes the command *string* to a command processor to be run. The command processor specified in the COMSPEC environment variable is first searched for. If it does not exist or is not a valid executable file, the default command processor, CMD.EXE, is searched for in the current directory and in all the directories specified in the PATH environment variable.

If the specified command is the name of an executable file created from a C program, full initialization and termination are performed, including automatic flushing of buffers and closing of files. To pass information across a system function, use a method of interprocess communication like pipes or shared memory.

You can also use system to redirect standard streams using the redirection operators (the angle brackets), for example:

```
rc = system("cprogram < in.file");
```

The defaults for the standard streams will be whatever the standard streams are at the point of the system call; for example, if the root program redirects **stdout** to file.txt, a printf call in a C module invoked by a system call will append to file.txt.

Return Value

If the argument is a null pointer, system returns nonzero if a command processor exists, and 0 if it does not. system returns the return value from the command processor if it is successfully called. If system cannot call the command processor successfully, the return value is -1 and errno is set to one of the following values:

Value	Meaning
ENOCMD	No valid command processor found.
ENOMEM	Insufficient memory to complete the function.
EOS2ERR	A system error occurred. Check _doserrno for the specific Windows error code.

system

This example shows how to use system to execute the command dir c:\.

```
#include <stdlib.h>

int main(void)
{
    int  rc;

     rc = system("dir c:\\");
    return  rc;

     /* The output should be a listing of the root directory on the c: drive       */
}
```

"exit — End Program" on page 179
"_exit — End Process" on page 180
"<process.h>" on page 772
"<stdlib.h>" on page 775

__sxchg — Exchange Integer Value with Memory

Format

```
#include <builtin.h>
short _Builtin __sxchg(volatile short*lockptr, short value);
```

Description

Language Level: Extension

__sxchg puts the specified *value* in the memory location pointed to by *lockptr*, and returns the value that was previously in that location.

Use this function to implement fast-RAM semaphores to serialize access to a critical resource (so that only one thread can use it at a time).

To implement a fast-RAM semaphore, allocate a volatile memory location *lockptr* (for __sxchg, it must be a short), and statically initialize it to 0 to indicate that the resource is free (not being used). To give a thread access to the resource, call __sxchg from the thread to exchange a non-zero value with that memory location. If __sxchg returns 0, the thread can access the resource; it has also set the memory location so that other threads can tell the resource is in use. When your thread no longer needs the resource, call __sxchg again to reset the memory location to 0.

If __sxchg returns a non-zero value, another thread is already using the resource, and the calling thread must wait for access using a Windows semaphore. You could then use the Windows semaphore to inform your waiting thread when the resource is unlocked by the thread currently using it.

It is important that you set the memory to a non-zero value when you are using the resource. You can use the same non-zero value for all threads, or a unique value for each.

Note: __sxchg is a built-in function, which means it is implemented as an inline instruction and has no backing code in the library. For this reason:

You cannot take the address of __sxchg.
You cannot parenthesize a call to __sxchg. (Parentheses specify a call to the function's backing code, and __sxchg has none.)

Return Value

__sxchg returns the previous value stored in the memory location pointed to by *lockptr*.

__sxchg

This example shows how __sxchg swaps two values.

```
#include <builtin.h>
#include <stdio.h>

void myswap (short *x, short *y ) {
     *x = __sxchg (y, *x);
}

int main (void) {
     short x, y;
    x=1;
    y=2;

     printf ("before swap, x=%d, y=%d\n", x, y);
     myswap (&x, &y);
     printf ("after swap, x=%d, y=%d\n", x, y);
    return  0;

    /*************************************************************************
         The output should be :

         before swap, x=1, y=2
         after swap, x=2, y=1
    *************************************************************************/
}
```

"__cxchg — Exchange Character Value with Memory" on page 119
"__lxchg — Exchange Integer Value with Memory" on page 369
"<builtin.h>" on page 761

tan — Calculate Tangent

Format

```
#include <math.h>
double tan(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

tan calculates the tangent of *x*, where *x* is expressed in radians. If *x* is too large, a partial loss of significance in the result can occur.

Return Value

tan returns the value of the tangent of *x*.

This example computes *x* as the tangent of π/4.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  pi,x;

    pi = 3.1415926;
    x = tan(pi/4.0);
    printf("tan( %lf ) is %lf\n", pi/4, x);
   return  0;

   /*****************************************************************************
        The output should be:

        tan( 0.785398 ) is 1.000000
   *****************************************************************************/
}
```

"atan – atan2 — Calculate Arctangent" on page 53
"cos — Calculate Cosine" on page 96
"_fpatan — Calculate Arctangent" on page 221
"_fptan — Calculate Tangent" on page 226
"sin — Calculate Sine" on page 514
"tanh — Calculate Hyperbolic Tangent" on page 618
"<math.h>" on page 770

tanh — Calculate Hyperbolic Tangent

Format

```
#include <math.h>
double tanh(double x);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

tanh calculates the hyperbolic tangent of *x*, where *x* is expressed in radians.

Return Value

tanh returns the value of the hyperbolic tangent of *x*. The result of tanh cannot have a range error.

This example computes *x* as the hyperbolic tangent of π/4.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double  pi,x;

    pi = 3.1415926;
    x = tanh(pi/4);
    printf("tanh( %lf ) = %lf\n", pi/4, x);
   return  0;

   /*****************************************************************************
        The output should be:

        tanh( 0.785398 ) = 0.655794

   *****************************************************************************/
}
```

"atan – atan2 — Calculate Arctangent" on page 53
"cosh — Calculate Hyperbolic Cosine" on page 97
"_fpatan — Calculate Arctangent" on page 221
"_fptan — Calculate Tangent" on page 226
"sinh — Calculate Hyperbolic Sine" on page 515
"tan — Calculate Tangent" on page 617
"<math.h>" on page 770

_tell — Get Pointer Position

Format

```
#include <io.h>
long _tell(int handle);
```

Description

Language Level: Extension

_tell gets the current position of the file pointer associated with *handle*. The position is the number of bytes from the beginning of the file.

Return Value

_tell returns the current position. A return value of −1L indicates an error, and errno is set to one of the following values:

Value	Meaning
EBADF	The file handle is not valid.
EOS2ERR	The call to the operating system was not successful.

On devices incapable of seeking (such as screens and printers), the return value is −1L.

This example opens the file tell.dat. It then calls _tell to get the current position of the file pointer and report whether the operation is successful. The program then closes tell.dat.

_tell

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#define FILENAME    "tell.dat"

int main(void)
{
    long  filePtrPos;
    int  fh;

    printf("Creating  file.\n");
     system("echo Sample Program > " FILENAME);
     if (-1 == (fh = open(FILENAME, O_RDWR|O_APPEND))) {
          perror("Unable to open " FILENAME);
          remove(FILENAME);
          return  EXIT_FAILURE;
    }
     /* Get the current file pointer position.                                    */
     if (-1 == (filePtrPos = _tell(fh))) {
          perror("Unable to tell");
          close(fh);
          remove(FILENAME);
          return  EXIT_FAILURE;
    }
     printf("File pointer is at position %d.\n", filePtrPos);
    close(fh);
    remove(FILENAME);
    return  0;

    /*************************************************************************
          The output should be:

          Creating  file.
          File pointer is at position 0.
    *************************************************************************/
}
```


"fseek — Reposition File Position" on page 247
"ftell — Get Current Position" on page 257
"lseek — Move File Pointer" on page 365
"<io.h>" on page 764

tempnam — Produce Temporary File Name

Format

```
#include <stdio.h>
char *tempnam(char *dir, char *prefix);
```

Description

Language Level: XPG4, Extension

tempnam creates a temporary file name in another directory. The *prefix* is the prefix to the file name. tempnam tests for the existence of the file with the given name in the following directories, listed in order of precedence:

If the TMP environment variable is set and the directory specified by TMP exists, the directory is specified by TMP.

If the TMP environment variable is not set or the directory specified by TMP does not exist, the directory is specified by the *dir* argument to tempnam.

If the *dir* argument is NULL, or *dir* is the name of nonexistent directory, the directory is pointed to by P_tmpdir (defined in <stdio.h>).

If P_tmpdir does not exist, the directory is the current working directory.

Notes:

1. Because tempnam uses malloc to reserve space for the created file name, you must free this space when you no longer need it.
2. In earlier releases of VisualAge C++, tempnam began with an underscore (_tempnam). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _tempnam to tempnam for you.

Return Value

tempnam returns a pointer to the temporary name, if successful. If the name cannot be created or it already exists, tempnam returns the value NULL.

This example creates a temporary file name using the directory d:\tmp:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  *name1;

    if ((name1 = tempnam("d:\\tmp", "stq")) != NULL)
        printf("%s is safe to use as a temporary file.\n", name1);
   else  {
        printf("Cannot create unique filename\n");
       return  EXIT_FAILURE;
   }
   return  0;

   /*************************************************************************
        The output should be similar to:

        D:\tmp\stqU3CP2.C2T is safe to use as a temporary file.
   *************************************************************************/
}
```

"malloc — Reserve Storage Block" on page 376
"_rmtmp — Remove Temporary Files" on page 482
"tmpfile — Create Temporary File" on page 626
"tmpnam — Produce Temporary File Name" on page 627
"<stdio.h>" on page 774

_threadstore — Access Thread-Specific Storage

Format

```
#include <stdlib.h>
void *_threadstore(void);
```

Description

Language Level: Extension

_threadstore provides access to a private thread pointer that is initialized to NULL. You can assign any thread-specific data structure to this pointer.

You can only use _threadstore in multithread programs (compiled with the /Gm+ option).

Return Value

_threadstore returns the address of the pointer to the defined thread storage space.

This example uses _threadstore to point to storage that is local to the thread. It prints the address pointed to by _threadstore.

Note: You must compile this example with the multithread option (/Gm+).

_threadstore

```
#if   (1 == __TOS_OS2__)
        #define  INCL_DOSPROCESS                    /*   For  OS/2   */
        #include  <os2.h>
        #define  _LNK_CONV    _Optlink
        #define  SLEEP    DosSleep(5000L)
#else
        #include  <windows.h>                       /*   For  Windows   */
        #define  _LNK_CONV
        #define  SLEEP    Sleep(5000L)
#endif

#include <stdlib.h>
#include <stdio.h>

#define   privateStore     (*_threadstore())

void _LNK_CONV thread(void *dummy)
{
     privateStore = malloc(100);
     printf("The starting address of the storaged space is %p\n", privateStore);

     /* user must free storage before exiting thread */
    free  (privateStore);
    _endthread();
}

int main(void)
{
    int  i;

     for (i = 0; i < 10; i++)
          _beginthread(thread, NULL, (unsigned) 32096, NULL);
    SLEEP;
    return  0;
}
```

"_beginthread — Create New Thread" on page 66
"_endthread — Terminate Current Thread" on page 169
"<stdlib.h>" on page 775

time — Determine Current Time

Format

```
#include <time.h>
time_t time(time_t *timeptr);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

time determines the current calendar time, which is not necessarily the local time localtime.

Note: The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Return Value

time returns the current calendar time. The return value is also stored in the location given by *timeptr*. If *timeptr* is NULL, the return value is not stored. If the calendar time is not available, the value (time_t)(-1) is returned.

This example gets the time and assigns it to *ltime*. ctime then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t  ltime;

   time(&ltime);
    printf("The time is %s\n", ctime(&ltime));
   return  0;

   /*****************************************************************************
        The output should be similar to:

        The time is Thu Jan 12 11:38:37 1995
   *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"ctime — Convert Time to Character String" on page 114
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"mktime — Convert Local Time" on page 410
"_strtime — Copy Time" on page 589
"<time.h>" on page 779

tmpfile — Create Temporary File

Format

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

tmpfile creates a temporary binary file. The file is automatically removed when it is closed or when the program is terminated.

tmpfile opens the temporary file in wb+ mode.

Return Value

tmpfile returns a stream pointer, if successful. If it cannot open the file, it returns a NULL pointer. On normal termination (exit), these temporary files are removed.

This example creates a temporary file, and if successful, writes tmpstring to it. At program termination, the file is removed.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
char tmpstring[] = "This is the string to be temporarily written";

int main(void)
{
    if (NULL == (stream = tmpfile())) {
        perror("Cannot make a temporary file");
        return  EXIT_FAILURE;
    }
    else
        fprintf(stream, "%s", tmpstring);
    return  0;
}
```

"fopen — Open Files" on page 218
"tmpnam — Produce Temporary File Name" on page 627
"tempnam — Produce Temporary File Name" on page 621
"_rmtmp — Remove Temporary Files" on page 482
"<stdio.h>" on page 774

tmpnam — Produce Temporary File Name

Format

```
#include <stdio.h>
char *tmpnam(char *string);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

tmpnam produces a valid file name that is not the same as the name of any existing file. It stores this name in *string*. If *string* is NULL, tmpnam leaves the result in an internal static buffer. Any subsequent calls destroy this value. If *string* is not NULL, it must point to an array of at least L_tmpnam bytes. The value of L_tmpnam is defined in <stdio.h>.

tmpnam produces a different name each time it is called within a module up to at least TMP_MAX (a value of at least 25) names. Note that files created using names returned by tmpnam are not automatically discarded at the end of the program. Files can be removed by the remove function.

Return Value

tmpnam returns a pointer to the name. If it cannot create a unique name; it returns NULL.

This example calls tmpnam to produce a valid file name.

```
#include <stdio.h>

int main(void)
{
   char  *name1;

    if ((name1 = tmpnam(NULL)) != NULL)
        printf("%s can be used as a file name.\n", name1);
   else
        printf("Cannot create a unique file name\n");

   return  0;

   /*************************************************************************
        The output should be similar to:

        d:\tmp\acc00000.CTN can be used as a file name.

   *************************************************************************/
}
```

"fopen — Open Files" on page 218
"remove — Delete File" on page 463
"tempnam — Produce Temporary File Name" on page 621
"<stdio.h>" on page 774

_toascii - _tolower - _toupper — Convert Character

Format

```
#include <ctype.h>
int _toascii(int c);
int _tolower(int c);
int _toupper(int c);
```

Description

Language Level: Extension

_toascii converts *c* to a character in the ASCII character set, by setting all but the low-order 7 bits to 0. If *c* already represents an ASCII character, _toascii does not change it.

_tolower converts *c* to the corresponding lowercase letter, if possible.

_toupper converts *c* to the corresponding uppercase letter, if possible.

Important: Use _tolower and _toupper only when you know that *c* is uppercase A to Z or lowercase a to z, respectively. Otherwise the results are undefined. These functions are not affected by the current locale.

These are all macros, and do not correctly handle arguments with side effects.

For portability, use the tolower and toupper functions defined by the ANSI/ISO standard, instead of the _tolower and _toupper macros.

Return Value

_toascii, _tolower, and _toupper return the possibly converted character *c*. If the character passed to _toascii is an ASCII character, _toascii returns the character unchanged. There is no error return.

This example prints four sets of characters. The first set is the ASCII characters having graphic images, which range from 0x21 through 0x7e. The second set takes integers 0x7f21 through 0x7f7e and applies the _toascii macro to them, yielding the same set of printable characters. The third set is the characters with all lowercase letters converted to uppercase. The fourth set is the characters with all uppercase letters converted to lowercase.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int  ch;

    printf("Characters 0x01 to 0x03, and integers 0x7f01 to 0x7f03 mapped to\n");
    printf("ASCII by _toascii() both yield the same graphic characters.\n\n");
    for (ch = 0x01; ch <= 0x03; ch++) {
        printf("char 0x%.4X: %c       ", ch, ch);
        printf("char _toascii(0x%.4X): %c\n", ch+0x7f00, ch+0x7f00);
   }
    printf("\nCharacters A, B and C converted to lower case and\n");
    printf("Characters a, b and c converted to upper case.\n\n");
    for (ch = 0x41; ch <= 0x43; ch++) {
        printf("_tolower(%c) = %c      ", ch, _tolower(ch));
        printf("_toupper(%c) = %c\n", ch+0x20, _toupper(ch+0x20));
   }
   return  0;

   /*****************************************************************************
        The output should be:

        Characters 0x01 to 0x03, and integers 0x7f01 to 0x7f03 mapped to
        ASCII by _toascii() both yield the same graphic characters.

        char 0x0001:        char _toascii(0x7F01):
        char 0x0002:        char _toascii(0x7F02):
        char 0x0003:        char _toascii(0x7F03):

        Characters A, B and C converted to lower case and
        Characters a, b and c converted to upper case.

        _tolower(A) = a     _toupper(a) = A
        _tolower(B) = b     _toupper(b) = B
        _tolower(C) = c     _toupper(c) = C
   *****************************************************************************/
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"_iscsym - _iscsymf — Test Integer" on page 325
"tolower - toupper — Convert Character Case" on page 631
"towlower – towupper — Convert Wide Character Case" on page 632

_toascii - _tolower - _toupper

"<ctype.h>" on page 762

tolower - toupper — Convert Character Case

Format

```
#include <ctype.h>
int tolower(int C);
int toupper(int c);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

tolower converts the uppercase letter *C* to the corresponding lowercase letter.

toupper converts the lowercase letter *c* to the corresponding uppercase letter.

The character mapping is determined by the LC_CTYPE category of the current locale.

Return Value

Both functions return the converted character. If the character *c* does not have a corresponding lowercase of uppercase character, the functions return *c* unchanged.

This example uses toupper and tolower to modify characters between code 0 and code 7f.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int  ch;

    /* print hex values of characters */
    for (ch = 0; ch <= 0x7f; ch++) {
        printf("toupper=%#04x\n",  toupper(ch));
        printf("tolower=%#04x\n",  tolower(ch));
        putchar('\n');
   }
   return  0;
}
```

"isalnum to isxdigit — Test Integer Value" on page 315
"isascii — Test Integer Values" on page 319
"_iscsym - _iscsymf — Test Integer" on page 325
"_toascii - _tolower - _toupper — Convert Character" on page 628
"towlower – towupper — Convert Wide Character Case" on page 632
"<ctype.h>" on page 762

towlower – towupper — Convert Wide Character Case

Format

```
#include <wctype.h>
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

Description

Language Level: ANSI 93, POSIX

towlower converts the uppercase letter *wc* to the corresponding lowercase letter.

towupper converts the lowercase letter *wc* to the corresponding uppercase letter.

The character mapping is determined by the LC_TIME category of current locale.

Return Value

Both functions return the converted character. If the wide character *wc* does not have a corresponding lowercase or uppercase character, the functions return *wc* unchanged.

This example uses towlower and towupper to convert characters between 0 and 0x7f.

```
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>

int main(void)
{
   wint_t  w_ch;

    for (w_ch = 0; w_ch <= 0x7f; w_ch++) {
        printf ("towupper : %#04x %#04x, ", w_ch, towupper(w_ch));
        printf ("towlower : %#04x %#04x\n", w_ch, towlower(w_ch));
   }
   return  0;

   /*************************************************************************
        The output should be similar to :
       .
       :
        towupper : 0x41 0x41, towlower : 0x41 0x61
        towupper : 0x42 0x42, towlower : 0x42 0x62
        towupper : 0x43 0x43, towlower : 0x43 0x63
        towupper : 0x44 0x44, towlower : 0x44 0x64
        towupper : 0x45 0x45, towlower : 0x45 0x65
       .
       :
        towupper : 0x61 0x41, towlower : 0x61 0x61
        towupper : 0x62 0x42, towlower : 0x62 0x62
        towupper : 0x63 0x43, towlower : 0x63 0x63
        towupper : 0x64 0x44, towlower : 0x64 0x64
        towupper : 0x65 0x45, towlower : 0x65 0x65
        :
   *************************************************************************/
}
```

"<wctype.h>" on page 782
"tolower - toupper — Convert Character Case" on page 631
"_toascii - _tolower - _toupper — Convert Character" on page 628

tzset — Assign Values to Locale Information

Format

```
#include <time.h>
void tzset(void);
```

Description

Language Level: XPG4, Extension

tzset uses the environment variable TZ to change the time zone and daylight saving time (DST) zone values of your current locale. These values are used by the gmtime and localtime functions to make corrections from Coordinated Universal Time (formerly GMT) to local time.

To use tzset, set the TZ variable to the appropriate values. (For the possible values for TZ, see the chapter on runtime environment variables in the *Programming Guide*.) Then call tzset to incorporate the changes in the time zone information into your current locale.

To set TZ from within a program, use putenv before calling tzset.

Notes:

1. In earlier releases of VisualAge C++, tzset began with an underscore (_tzset). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _tzset to tzset for you.
2. The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Return Value There is no return value.

This example uses putenv and tzset to set the time zone to Central Time.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    time_t  currentTime;
     struct tm *ts;

     /* Get the current time                                                    */

    (void)time(&currentTime);
     printf("The GMT time is %s", asctime(gmtime(&currentTime)));
     ts = localtime(&currentTime);
     if (ts->tm_isdst > 0)    /* check if Daylight Saving Time is in effect     */
         {
         printf("The local time is %s", asctime(ts));
         printf("Daylight Saving Time is in effect.\n");
    }
    else  {
         printf("The local time is %s", asctime(ts));
         printf("Daylight Saving Time is not in effect.\n");
    }
     printf("**** Changing to Central Time ****\n");
    putenv("TZ=CST6CDT");
    tzset();
     ts = localtime(&currentTime);
     if (ts->tm_isdst > 0)    /* check if Daylight Saving Time is in effect     */
         {
         printf("The local time is %s", asctime(ts));
         printf("Daylight Saving Time is in effect.\n");
    }
    else  {
         printf("The local time is %s", asctime(ts));
         printf("Daylight Saving Time is not in effect.\n");
    }

    return  0;

    /*****************************************************************************
         The output should be similar to:

         The GMT time is Fri Jan 13 21:49:26 1995
         The local time is Fri Jan 13 16:49:26 1995
         Daylight Saving Time is not in effect.
         **** Changing to Central Time ****
         The local time is Fri Jan 13 15:49:26 1995
         Daylight Saving Time is not in effect.
    *****************************************************************************/
}
```

"asctime — Convert Time to Character String" on page 47
"_ftime — Store Current Time" on page 259
"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356

"mktime — Convert Local Time" on page 410
"putenv — Modify Environment Variables" on page 439
"strftime — Convert to Formatted Time" on page 557
"time — Determine Current Time" on page 625
"<time.h>" on page 779

_uaddmem — Add Memory to a Heap

Format

#include <umalloc.h>
Heap_t _uaddmem(Heap_t *heap*, void **block*, size_t *size*, int *clean*);

Description

Language Level: Extension

_uaddmem adds a *block* of memory of *size* bytes into the specified user *heap* (created with _ucreate). Before calling _uaddmem, you must first get the *block* from the operating system, typically by using an Win32 API like VirtualAlloc or by allocating it statically. (See the *Win32 Programmer's Reference* for details on Win32 APIs for memory management.)

If the memory *block* has been initialized to 0, specify _BLOCK_CLEAN for the *clean* parameter. If not, specify !_BLOCK_CLEAN. (This information makes calloc and _ucalloc more efficient).

Note: Memory returned by VirtualAlloc is initialized to 0.

For fixed-size heaps, you must return all the blocks you added with _uaddmem to the system. (For expandable heaps, these blocks are returned by your *release_fn* when you call _udestroy.)

For more information about creating and using heaps, see "Managing Memory" in the *Programming Guide*.

Note: For every block of memory you add, a small number of bytes from it are used to store internal information. To reduce the total amount of overhead, it is better to add a few large blocks of memory than many small blocks.

Return Value

_uaddmem returns a pointer to the heap the memory was added to. If the heap specified is not valid, _uaddmem returns NULL.

The following example creates a heap myheap, and then uses _uaddmem to add memory to it.

_uaddmem

```
#if   (1 == __TOS_OS2__)                    /*   For OS/2   */
    #define   INCL_DOSMEMMGR                    /* Memory Manager values */
   #include  <os2.h>
    #include <bsememf.h>                    /* Get flags for memory management   */
#else
   #include  <windows.h>                    /*   For  Windows   */
#endif

#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   void       *initial_block,  *extra_chunk;
   Heap_t    myheap;
   char       *p1,  *p2;

#if   (1 == __TOS_OS2__)
   APIRET    rc;

    /* Call DosAllocMem to get the initial block of memory */
    if (0 != (rc = DosAllocMem(&initial_block, 65536,
                                        PAG_WRITE | PAG_READ | PAG_COMMIT)))
   {
        printf("DosAllocMem for initial block failed: return code = %ld\n", rc);

#else
    /* Call VirtualAlloc to get the initial block of memory */
    if (NULL == (initial_block =
          VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
   {
        printf("VirtualAlloc for initial block failed: %d.\n", GetLastError());
```

```
#endif
        exit(EXIT_FAILURE);
    }
    /* Create a fixed size heap starting with the block declared earlier */
    if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                                _HEAP_REGULAR, NULL, NULL))) {
        puts("_ucreate  failed.");
        exit(EXIT_FAILURE);
    }
    if (0 != _uopen(myheap)) {
        puts("_uopen  failed.");
        exit(EXIT_FAILURE);
    }
    p1 = (char*)_umalloc(myheap, 100);
#if   (1 == __TOS_OS2__)
    /* Call DosAllocMem to get another block of memory */
    if (0 != (rc = DosAllocMem(&extra_chunk, 10 * 65536,
                                        PAG_WRITE | PAG_READ | PAG_COMMIT)))
    {
         printf("DosAllocMem for extra chunk failed: return code = %ld\n", rc);

#else
    /* Call VirtualAlloc to get another block of memory */
    if (NULL == (extra_chunk =
          VirtualAlloc(NULL, 10 * 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
    {
         printf("VirtualAlloc for extra chunk failed: %d.\n", GetLastError());

#endif
        exit(EXIT_FAILURE);
    }
    /* Add the second chunk of memory to user heap */
    if (myheap != _uaddmem(myheap, extra_chunk, 10 * 65536, _BLOCK_CLEAN)) {
        puts("_uaddmem  failed.");
        exit(EXIT_FAILURE);
    }
    p2 = (char*)_umalloc(myheap, 100000);
   free(p1);
   free(p2);
    if (0 != _uclose(myheap)) {
        puts("_uclose  failed");
        exit(EXIT_FAILURE);
    }
#if   (1 == __TOS_OS2__)
    if (0 != DosFreeMem(initial_block) || 0 != DosFreeMem(extra_chunk))
    {
        puts("DosFreeMem  error.");
```

```
#else
    if (FALSE == VirtualFree(initial_block, 0, MEM_RELEASE) ||
          FALSE == VirtualFree(extra_chunk, 0, MEM_RELEASE))
    {
        puts("VirtualFree  error.");

#endif
        exit(EXIT_FAILURE);
    }
    return  0;
}
```

"_ucreate — Create a Memory Heap" on page 646
"_udestroy — Destroy a Heap" on page 653
"_uheapmin — Release Unused Memory in User Heap" on page 667
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_ucalloc — Reserve and Initialize Memory from User Heap

Format

```
#include <umalloc.h>
void *_ucalloc(Heap_t heap, size_t num, size_t size);
```

Description

Language Level: Extension

_ucalloc allocates memory for an array of *num* elements, each of length *size* bytes, from the *heap* you specify. It then initializes all bits of each element to 0.

_ucalloc works just like calloc except that you specify the heap to use; calloc always allocates from the default heap. A debug version of this function, _debug_ucalloc, is also provided.

If the *heap* does not have enough memory for the request, _ucalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _ucalloc, use the non-heap-specific realloc and free. These functions always check what heap the memory was allocated from.

Return Value

_ucalloc returns a pointer to the reserved space. If *size* or *num* was specified as zero, or if your *getmore_fn* cannot provide enough memory, _ucalloc returns NULL. Passing _ucalloc a heap that is not valid results in undefined behavior.

This example creates a heap myheap and then uses _ucalloc to allocate memory from it.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
       exit(EXIT_FAILURE);
   }
    memset(ptr, 'x', 10);
   free(ptr);
   return  0;
}
```

_ucalloc

"calloc — Reserve and Initialize Storage" on page 75
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"free — Release Storage Blocks" on page 238
"realloc — Change Reserved Storage Block Size" on page 452
"_ucreate — Create a Memory Heap" on page 646
"_umalloc — Reserve Memory Blocks from User Heap" on page 677
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_uclose — Close Heap from Use

Format

```
#include <umalloc.h>
int _uclose(Heap_t heap);
```

Description

Language Level: Extension

_uclose closes a *heap* when a process will not use it again. After you close a heap, any attempt in the current process to allocate or return memory to it will have undefined results. _uclose affects only the current process; if the heap is shared, other processes may still be able to access it.

Once you have closed the heap, use _udestroy to destroy it and return all its memory to the operating system.

Note: If the heap is shared, you must close it in all processes that share it before you destroy it, or undefined results will occur.

You cannot close the VisualAge for C++ runtime heap (_RUNTIME_HEAP).

For more information about creating and using heaps, see "Managing Memory" in the *Programming Guide*.

Return Value

If successful, _uclose returns 0. A nonzero return value indicates failure. Passing _uclose a heap that is not valid results in undefined behavior.

This example creates and opens a heap, and then performs operations on it. It then calls _uclose to close the heap before destroying it.

```
#if   (1 == __TOS_OS2__)                    /*   For OS/2   */
       #define   INCL_DOSMEMMGR                    /* Memory Manager values */
      #include  <os2.h>
       #include <bsememf.h>                 /* Get flags for memory management   */
#else
      #include  <windows.h>                 /*   For  Windows   */
#endif

#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>
```

_uclose

```
int main(void)
{
    void       *initial_block;
    Heap_t     myheap;
    char       *p;

#if    (1 == __TOS_OS2__)
    APIRET     rc;

     /* Call DosAllocMem to get the initial block of memory */
     if (0 != (rc = DosAllocMem(&initial_block, 65536,
                                        PAG_WRITE | PAG_READ | PAG_COMMIT)))
    {
         printf("DosAllocMem error: return code = %ld\n", rc);

#else
     /* Call VirtualAlloc to get the initial block of memory */
     if (NULL == (initial_block =
           VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
    {
         printf("VirtualAlloc error: %d\n", GetLastError());

#endif
         exit(EXIT_FAILURE);
    }
     /* Create a fixed size heap starting with the block declared earlier */
     if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                                 _HEAP_REGULAR, NULL, NULL))) {
         puts("_ucreate  failed.");
         exit(EXIT_FAILURE);
    }
     if (0 != _uopen(myheap)) {
         puts("_uopen  failed.");
         exit(EXIT_FAILURE);
    }
     p = (char *)_umalloc(myheap, 100);
     memset(p, 'x', 10);
    free(p);

     if (0 != _uclose(myheap)) {
         puts("_uclose  failed");
         exit(EXIT_FAILURE);
    }
```

```
#if    (1 == __TOS_OS2__)
     if (0 != (rc = DosFreeMem(initial_block)))
     {
          printf("DosFreeMem error: return code = %ld\n", rc);

#else
     if (FALSE == (VirtualFree(initial_block, 0, MEM_RELEASE)))
     {
          printf("VirtualFree error: %d.\n", GetLastError());

#endif
          exit(EXIT_FAILURE);
     }
    return  0;
}
```

"_uopen — Open Heap for Use" on page 689
"_udestroy — Destroy a Heap" on page 653
"_ucreate — Create a Memory Heap" on page 646
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_ucreate — Create a Memory Heap

Format

```
#include <umalloc.h>
Heap_t _ucreate(void *block, size_t initsz, int clean, int memtype,
                void *(*getmore_fn)(Heap_t, size_t *, int *)
                void   (*release_fn)(Heap_t, void *, size_t);
```

Description

Language Level: Extension

_ucreate creates your own memory heap that you can allocate and free memory from, just like the VisualAge for C++ runtime heap.

Before you call _ucreate, you must first get the initial *block* of memory for the heap. You can get this block by calling a Win32 API (such as VirtualAlloc or CreateFileMapping) or by statically allocating it. (See the *Win32 Programmer's Reference* for more information on the Win32 APIs.) If required by the application, shared memory must be explicitely mapped by you to the same address in all processes sharing the memory. When the heaps are user-defined, you must explicitely map shared memory.

Note: You must also return this initial block of memory to the system after you destroy the heap.

When you call _ucreate, you pass it the following parameters:

block The pointer to the initial block you obtained.

initsz The size of the initial block, which must be at least _HEAP_MIN_SIZE bytes (defined in <malloc.h>). If you are creating a fixed-size heap, the size must be large enough to satisfy all memory requests your program will make of it.

clean The macro _BLOCK_CLEAN, if the memory in the block has been initialized to 0, or !_BLOCK_CLEAN, if the memory has not been touched. This improves the efficiency of _ucalloc; if the memory is already initialized to 0, _ucalloc does not need to initialize it.

Note: VirtualAlloc initializes memory to 0 for you. You can also use memset to initialize the memory; however, memset also commits all the memory at once, an action that could slow overall performance.

memtype A macro indicating the type of memory in your heap: _HEAP_REGULAR (regular). If you want the memory to be shared, specify _HEAP_SHARED as well (for example, _HEAP_REGULAR | _HEAP_SHARED). Shared memory can be shared

between different processes. For more information on different types of memory, see the *Programming Guide*.

Note: Make sure that when you get the initial block, you request the same type of memory that you specify for *memtype*.

getmore_fn A function you provide to get more memory from the system (typically using Windows APIs or static allocation). To create a fixed-size heap, specify NULL for this parameter.

release_fn A function you provide to return memory to the system. To create a fixed-size heap, specify NULL for this parameter.

If you create a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. Once the block is fully allocated, further allocation requests to the heap will fail. If you create an expandable heap, the *getmore_fn* and *release_fn* allow your heap to expand and shrink dynamically.

When you call _umalloc (or a similar function) for your heap, if not enough memory is available in the block, it calls the *getmore_fn* you provide. Your *getmore_fn* then gets more memory from the system and adds it to the heap, using any method you choose.

Your *getmore_fn* must have the following prototype:

```
void *(*getmore_fn)(Heap_t uh, size_t *size, int *clean);
```

where:

uh Is the heap to get memory for.

size Is the size of the allocation request passed by _umalloc. You probably want to return enough memory at a time to satisfy several allocations; otherwise, every subsequent allocation has to call *getmore_fn*. You should return multiples of 64K (the smallest size that VirtualAlloc returns). Make sure you update the *size* parameter if you return more than the original request.

clean Within *getmore_fn*, you must set this variable either to _BLOCK_CLEAN, to indicate that you initialized the memory to 0, or to !_BLOCK_CLEAN, to indicate that the memory is untouched.

Note: Make sure your *getmore_fn* allocates the right type of memory for the heap.

When you call _uheapmin to coalesce the heap or _udestroy to destroy it, these functions call the *release_fn* you provide to return the memory to the system.

Your *release_fn* must have the following prototype:

```
void (*release_fn)(Heap_t uh, void *block, size_t size);
```

The heap *uh*, the *block* to be returned, and its *size* are passed to *release_fn* by _uheapmin or _udestroy.

For more information about creating and using heaps, see the "Managing Memory" in the *Programming Guide*.

Return Value

If successful, _ucreate returns a pointer to the heap created. If errors occur, _ucreate returns NULL.

This example uses _ucreate to create an expandable heap. The functions for expanding and shrinking the heap are get_fn and release_fn. The program then opens the heap and performs operations on it, and then closes and destroys the heap.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void* get_fn(Heap_t usrheap, size_t *length, int *clean)
{
     void  *p;

      /* Round up to the next chunk size */
      *length = ((*length) / 65536) * 65536 + 65536;
      *clean = _BLOCK_CLEAN;
      p = VirtualAlloc(NULL, *length, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
     return  (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
      VirtualFree(p, 0, MEM_RELEASE );
     return;
}

int main(void)
{
```

```
    void       *initial_block;
    Heap_t     myheap;
    char       *ptr;

    /* Call VirtualAlloc to get the initial block of memory */
    if (NULL == (initial_block =
          VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
    {
         printf("VirtualAlloc error: %d.\n", GetLastError());
        exit(EXIT_FAILURE);
    }
    /* Create an expandable heap starting with the block declared earlier */
    if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                          _HEAP_REGULAR, get_fn, release_fn))) {
        puts("_ucreate  failed.");
        exit(EXIT_FAILURE);
    }
    if (0 != _uopen(myheap)) {
        puts("_uopen  failed.");
        exit(EXIT_FAILURE);
    }

    /* Force user heap to grow */
    ptr = (char *)_umalloc(myheap, 100000);

    free(ptr);

    _uclose(myheap);

    if (0 != _udestroy(myheap, _FORCE)) {
        puts("_udestroy  failed.");
        exit(EXIT_FAILURE);
    }
    if (FALSE == VirtualFree(initial_block, 0, MEM_RELEASE)) {
         printf("VirtualFree error: %d.\n", GetLastError());
        exit(EXIT_FAILURE);
    }
    return  0;
}
```

"_uaddmem — Add Memory to a Heap" on page 637
"_ucalloc — Reserve and Initialize Memory from User Heap" on page 641
"_uclose — Close Heap from Use" on page 643
"_udestroy — Destroy a Heap" on page 653
"_uheapmin — Release Unused Memory in User Heap" on page 667
"_umalloc — Reserve Memory Blocks from User Heap" on page 677
"_uopen — Open Heap for Use" on page 689
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_udefault — Change the Default Heap

Format

#include <umalloc.h>
Heap_t _udefault(Heap_t *heap*);

Description

Language Level: Extension

_udefault makes the *heap* you specify become the default heap. All calls to memory management functions that do not specify a heap (including malloc and calloc) then allocate memory from the *heap*.

This change affects only the thread where you called _udefault.

The initial default heap is the VisualAge for C++ runtime heap. To restore or explicitly set the VisualAge for C++ runtime heap as the default, call _udefault with the argument _RUNTIME_HEAP.

You can also use _udefault to find out which heap is being used as the default by specifying NULL for the *heap* parameter. The default heap remains the same.

For more information about creating and using heaps, see "Managing Memory" in the *Programming Guide*.

Return Value

_udefault returns a pointer to the former default heap. You can save this pointer and use it later to restore the original heap. If the call is unsuccessful, _udefault returns NULL. Passing _udefault a heap that is not valid results in undefined behavior.

This example creates a fixed-size heap myheap and uses _udefault to make it the default heap. The call to malloc then allocates memory from myheap. The second call to _udefault restores the original default heap.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
    void       *initial_block;
    Heap_t     myheap,  old_heap;
    char       *p;

     /* Call VirtualAlloc to get the initial block of memory */
     if (NULL == (initial_block =
            VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
    {
          printf("VirtualAlloc error: %d\n.", GetLastError());
         exit(EXIT_FAILURE);
    }
     /* Create a fixed size heap starting with the block declared earlier */
     if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                                 _HEAP_REGULAR, NULL, NULL))) {
         puts("_ucreate  failed.");
         exit(EXIT_FAILURE);
    }
     if (0 != _uopen(myheap)) {
         puts("_uopen  failed.");
         exit(EXIT_FAILURE);
    }

     /* myheap is used as default heap */
     old_heap = _udefault(myheap);

     /* malloc will allocate memory from myheap */
     p = (char *)malloc(100);
     memset(p, 'x', 10);

     /* Restore original default heap */
    _udefault(old_heap);

    free(p);
     if (0 != _uclose(myheap)) {
         puts("_uclose  failed");
         exit(EXIT_FAILURE);
    }
     if (FALSE == VirtualFree(initial_block, 0, MEM_RELEASE)) {
          printf("VirtualFree error: %d\n.", GetLastError());
         exit(EXIT_FAILURE);
    }
    return  0;
}
```

"calloc — Reserve and Initialize Storage" on page 75
"malloc — Reserve Storage Block" on page 376
"_mheap — Query Memory Heap for Allocated Object" on page 404
"_ucreate — Create a Memory Heap" on page 646

_udefault

"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_udestroy — Destroy a Heap

Format

#include <umalloc.h>
int _udestroy(Heap_t *heap*, int *force*);

Description

Language Level: Extension

_udestroy destroys the *heap* you specify. It also returns the heap's memory to the system by calling the *release_fn* you supplied to _ucreate when you created the heap. If you did not supply a *release_fn*, _udestroy simply marks the heap as destroyed so no further operations can be performed. You must then return all the memory in the heap to the system.

Note: Whether or not you provide a *release_fn*, you must always return the initial block of memory (that you provided to _ucreate) to the system.

The *force* parameter controls the behavior of _udestroy if all allocated objects from the heap have not been freed. If you specify _FORCE for this parameter, _udestroy destroys the heap regardless of whether allocated objects remain in that process or in any other process that shares the heap. If you specify !_FORCE, the heap will not be destroyed if any objects are still allocated from it.

Typically, you call _uclose to close the heap before you destroy it. After you have destroyed a heap, any attempt to access it will have undefined results.

You cannot destroy the VisualAge for C++ runtime heap (_RUNTIME_HEAP).

Return Value

_udestroy returns 0 whether the heap was destroyed or not. If the heap passed to it is not valid, _udestroy returns a nonzero value.

This example creates and opens a heap, performs operations on it, and then closes it. The program then calls _udestroy with the _FORCE parameter to force the destruction of the heap. _udestroy calls release_fn to return the memory to the system.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
```

_udestroy

```
static void* get_fn(Heap_t usrheap, size_t *length, int *clean)
{
      void  *p;

       /* Round up to the next chunk size */
       *length = ((*length) / 65536) * 65536 + 65536;
       *clean = _BLOCK_CLEAN;
       p = VirtualAlloc(NULL, *length, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
      return  (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
       VirtualFree(p, 0, MEM_RELEASE );
      return;
}

int main(void)
{
    void       *initial_block;
    Heap_t    myheap;
    char       *ptr;

     /* Call VirtualAlloc to get the initial block of memory */
     if (NULL == (initial_block =
           VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
    {
          printf("VirtualAlloc error: %d.\n", GetLastError());
         exit(EXIT_FAILURE);
    }
     /* Create an expandable heap starting with the block declared earlier */
     if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                                 _HEAP_REGULAR, get_fn, release_fn))) {
         puts("_ucreate  failed.");
         exit(EXIT_FAILURE);
    }
     if (0 != _uopen(myheap)) {
         puts("_uopen  failed.");
         exit(EXIT_FAILURE);
    }
```

```
    /* Force user heap to grow */
    ptr = (char *)_umalloc(myheap, 100000);

    free(ptr);

    _uclose(myheap);

    if (0 != _udestroy(myheap, _FORCE)) {
        puts("_udestroy  failed.");
        exit(EXIT_FAILURE);
    }
    if (FALSE == VirtualFree(initial_block, 0, MEM_RELEASE)) {
         printf("VirtualFree error: %d.\n", GetLastError());
        exit(EXIT_FAILURE);
    }
    return  0;
}
```

"_uaddmem — Add Memory to a Heap" on page 637
"_ucreate — Create a Memory Heap" on page 646
"_uopen — Open Heap for Use" on page 689
"_uclose — Close Heap from Use" on page 643
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_udump_allocated — Get Information about Allocated Memory in Heap

Format

#include <umalloc.h>
void _udump_allocated(Heap_t *heap*, int *nbytes*);

Description

Language Level: Extension

For the *heap* you specify, _udump_allocated prints information to **stderr** about each memory block that is currently allocated and was allocated using the debug memory management functions (_debug_ucalloc, _debug_umalloc, and so on).

_udump_allocated works just like _dump_allocated, except that you specify the heap to use; _dump_allocated always displays information about the default heap.

Use *nbytes* to specify how many bytes of each memory block are to be printed. If *nbytes* is:

Negative value	Prints all bytes of each block.
0	Prints no bytes.
Positive value	Prints the specified number of bytes or the entire block, whichever is smaller.

_udump_allocated prints the information to **stderr** by default. You can change the destination with the _set_crt_msg_handle function.

Call _udump_allocated at points in your code where you want a report of the currently allocated memory. For example, a good place to call _udump_allocated is a point where most of the memory is already freed and you want to find a memory leak, such as at the end of a program.

You can also use _udump_allocated_delta to display information about only the memory that was allocated since the previous call to _udump_allocated or _udump_allocated_delta.

To use _udump_allocated and the debug versions of the memory management functions, specify the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value There is no return value. Passing _udump_allocated a heap that is not valid results in undefined behavior.

This example creates a heap, performs some operations on it, and then calls _udump_allocated to print out information about the allocated memory blocks.

Note: You must compile this example with the /Tm option to enable the debug memory management functions.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_umalloc(myheap, 10))) {
        puts("Cannot allocate memory from user heap.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'a', 5);

   _udump_allocated(myheap,  10);

   free(ptr);
   return  0;

   /*************************************************************************
        The output should be similar to :

       ---------------------------------------------------------------------
                              START OF DUMP OF ALLOCATED MEMORY BLOCKS
       ---------------------------------------------------------------------
        Address: 0x00073890          Size: 0x0000000A (10)
        This memory block was (re)allocated at line number 14 in _udump_alloc.c.
        Memory contents:   61616161 61AAAAAA AAAA                    [aaaaaêêêêê          ]
       ---------------------------------------------------------------------
                              END OF DUMP OF ALLOCATED MEMORY BLOCKS
       ---------------------------------------------------------------------
   *************************************************************************/
}
```

"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"_debug_realloc — Reallocate Memory Block" on page 132
"_debug_free — Release Memory" on page 126
"_dump_allocated — Get Information about Allocated Memory" on page 154

_udump_allocated

"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_set_crt_msg_handle — Change Runtime Message Output Destination" on page 495
"_udump_allocated_delta — Get Information about Allocated Memory in Heap" on page 659
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_udump_allocated_delta — Get Information about Allocated Memory in Heap

Format

#include <umalloc.h>
void _udump_allocated_delta(Heap_t *heap*, int *nbytes*);

Description

Language Level: Extension

For the *heap* you specify, _udump_allocated_delta prints information to **stderr** about each memory block allocated by a debug memory management function (_debug_umalloc and so on) since the last call to _udump_allocated_delta or _udump_allocated.

_udump_allocated_delta and _udump_allocated print the same type of information to **stderr**, but _udump_allocated displays information about all blocks that have been allocated since the start of your program.

_udump_allocated_delta works just like _dump_allocated_delta, except that you specify the heap to use; _dump_allocated_delta always displays information about the default heap.

Use *nbytes* to specify how many bytes of each memory block are to be printed. If *nbytes* is:

Negative value	Prints all bytes of each block.
0	Prints no bytes.
Positive value	Prints the specified number of bytes or the entire block, whichever is smaller.

_udump_allocated_delta prints the information to **stderr** by default. You can change the destination with the _set_crt_msg_handle function.

To use _udump_allocated_delta and the debug versions of the memory management functions, specify the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value

There is no return value. Passing _udump_allocated_delta a heap that is not valid results in undefined behavior.

_udump_allocated_delta

This example creates a heap and allocates memory from it. It then calls _udump_allocated to display information about the allocated memory. After performing more memory operations, it calls _udump_allocated_delta to display information about the memory allocated since the call to _udump_allocated

Note: You must compile this example with the /Tm option to enable the debug memory management functions.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
    Heap_t    myheap;
    char      *ptr1,  *ptr2;

     /* Use default heap as user heap */
     myheap = _udefault(NULL);

     if (NULL == (ptr1 = (char*)_umalloc(myheap, 10))) {
          puts("Cannot allocate first memory block from user heap.");
          exit(EXIT_FAILURE);
     }
     memset(ptr1, 'a', 5);
    _udump_allocated(myheap,  10);

     if (NULL == (ptr2 = (char*)_umalloc(myheap, 20))) {
          puts("Cannot allocate second memory block from user heap.");
          exit(EXIT_FAILURE);
     }
     memset(ptr2, 'b', 5);
     printf("\nResults of _udump_allocated_delta are:\n");
    _udump_allocated_delta(myheap,  10);

    free(ptr1);
    free(ptr2);
    return  0;
```

```
/*****************************************************************************
    The output should be similar to :

    ---------------------------------------------------------------------------
                        START OF DUMP OF ALLOCATED MEMORY BLOCKS
    ---------------------------------------------------------------------------
    Address: 0x00073890         Size: 0x0000000A (10)
    This memory block was (re)allocated at line number 14 in _udump_alloc_d.c.
    Memory contents:   61616161 61AAAAAA AAAA                   [aaaaaêêêêê         ]
    ---------------------------------------------------------------------------
                        END OF DUMP OF ALLOCATED MEMORY BLOCKS
    ---------------------------------------------------------------------------

    Results of _udump_allocated_delta are:
    ---------------------------------------------------------------------------
                        START OF DUMP OF ALLOCATED MEMORY BLOCKS
    ---------------------------------------------------------------------------
    Address: 0x000738D0         Size: 0x00000014 (20)
    This memory block was (re)allocated at line number 21 in _udump_alloc_d.c.
    Memory contents:   62626262 62AAAAAA AAAA                   [bbbbbêêêêê         ]
    ---------------------------------------------------------------------------
                        END OF DUMP OF ALLOCATED MEMORY BLOCKS
    ---------------------------------------------------------------------------
*****************************************************************************/
}
```

"_dump_allocated — Get Information about Allocated Memory" on page 154
"_dump_allocated_delta — Get Information about Allocated Memory" on page 157
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"_debug_realloc — Reallocate Memory Block" on page 132
"_set_crt_msg_handle — Change Runtime Message Output Destination" on page 495
"_udump_allocated — Get Information about Allocated Memory in Heap" on page 656
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_uheap_check — Validate User Memory Heap

Format

```
#include <umalloc.h>
void _uheap_check(Heap_t heap);
```

Description

Language Level: Extension

_uheap_check checks all memory blocks in the *heap* you specify that have been allocated or freed using the heap-specific debug versions of the memory management functions (_debug_ucalloc, _debug_umalloc, and so on). _uheap_check checks that your program has not overwritten freed memory blocks or memory outside the bounds of allocated blocks.

_uheap_check works just like _heap_check, except that you specify the heap to check; _heap_check always checks the default heap.

When you call a heap-specific debug memory management function (such as _debug_umalloc), it calls _uheap_check automatically. You can also call _uheap_check explicitly. Place calls to _uheap_check throughout your code, especially in areas that you suspect have memory problems.

Calling _uheap_check frequently can increase your program's memory requirements and decrease its execution speed. To reduce the overhead of heap-checking, you can use the CPP_HEAP_SKIP environment variable to control how often the functions check the heap. For example:

```
SET CPP_HEAP_SKIP=10
```

specifies that every tenth call to any debug memory function (regardless of the type or heap) checks the heap. Explicit calls to _uheap_check are always performed. For more details on CPP_HEAP_SKIP, see “Skipping Heap Checks” in the *Programming Guide*.

To use _uheap_check and the debug versions of the memory management functions, specify the debug memory (/Tm) compiler option.

Note: The /Tm option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value There is no return value.

This example creates a heap and allocates memory from it. It then calls _uheap_check to check the the memory in the heap.

Note: You must compile this example with the /Tm option to map the _ucalloc calls to _debug_ucalloc.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
       exit(EXIT_FAILURE);
   }
    memset(ptr, 'x', 105);        /* overwrite storage that was not allocated */

   _uheap_check(myheap);

   free(ptr);
   return  0;

   /************************************************************************
        The output should be similar to :

        End of allocated object 0x00073890 was overwritten at 0x000738f4.
        The first eight bytes of the memory block (in hex) are: 7878787878787878.
        This memory block was (re)allocated at line number 13 in _uheap_check.c.
        Heap state was valid at line 13 of _uheap_check.c.
        Memory error detected at line 19 of _uheap_check.c.
   ************************************************************************/
}
```

"_heap_check — Validate Default Memory Heap" on page 291
"_debug_ucalloc — Reserve and Initialize Memory from User Heap" on page 134
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"_debug_realloc — Reallocate Memory Block" on page 132
"_debug_free — Release Memory" on page 126
"_uheapchk — Validate Memory Heap" on page 665
"_uheapset — Validate and Set Memory Heap" on page 669
"_uheap_walk — Return Information about Memory Heap" on page 671
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*

_uheap_check

"<umalloc.h>" on page 779

_uheapchk — Validate Memory Heap

Format

#include <umalloc.h>
int _uheapchk(Heap_t *heap*);

Description

Language Level: Extension

_uheapchk checks the *heap* you specify for minimal consistency by checking all allocated and freed objects on the heap.

_uheapchk works just like _heapchk, except that you specify the heap to check; _heapchk always checks the default heap.

Note: Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap.

Return Value

_uheapchk returns one of the following values, defined in both <umalloc.h> and <malloc.h>:

_HEAPBADBEGIN	The heap specifed is not valid. It may have been closed or destroyed.
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

This example creates a heap and performs memory operations on it. It then calls _uheapchk to validate the heap.

_uheapchk

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t    myheap;
   char      *ptr;
   int       rc;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
        exit(EXIT_FAILURE);
    }
    *(ptr - 1) = 'x';         /* overwrite storage that was not allocated */

    if (_HEAPOK != (rc = _uheapchk(myheap))) {
        switch(rc)  {
            case  _HEAPEMPTY:
                  puts("The heap has not been initialized.");
                  break;
            case  _HEAPBADNODE:
                  puts("A memory node is corrupted or the heap is damaged.");
                  break;
            case  _HEAPBADBEGIN:
                  puts("The heap specified is not valid.");
                  break;
        }
        exit(rc);
   }
   free(ptr);
   return  0;

   /*************************************************************************
        The output should be similar to :

        A memory node is corrupted or the heap is damaged.
   *************************************************************************/
}
```

"_heapchk — Validate Default Memory Heap" on page 293
"_heapmin — Release Unused Memory from Default Heap" on page 295
"_uheapset — Validate and Set Memory Heap" on page 669
"_uheap_walk — Return Information about Memory Heap" on page 671
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_uheapmin — Release Unused Memory in User Heap

Format

```
#include <umalloc.h>
int _uheapmin(Heap_t heap);
```

Description

Language Level: Extension

_uheapmin returns all unused memory blocks from the *heap* you specify to the operating system.

_uheapmin works just like _heapmin, except that you specify the heap to use; _heapmin always uses the default heap. A debug version of this function, _debug_uheapmin, is also provided.

To return the memory, _uheapmin calls the *release_fn* you supplied when you created the heap with _ucreate. If you did not supply a *release_fn*, _uheapmin has no effect and simply returns 0.

Return Value

If successful, _uheapmin returns 0. A nonzero return value indicates failure. Passing _uheapmin a heap that is not valid has undefined results.

This example creates a heap and then allocates and frees a large block of memory from it. It then calls _uheapmin to return free blocks of memory to the system.

_uheapmin

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
    Heap_t    myheap;
    char      *ptr;

     /* Use default heap as user heap */
     myheap = _udefault(NULL);

     /* Allocate a large object */
     if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
          puts("Cannot allocate memory from user heap.");
          exit(EXIT_FAILURE);
     }
     memset(ptr, 'x', 60000);
    free(ptr);

     /* _uheapmin will attempt to return the freed object to the system */
     if (0 != _uheapmin(myheap)) {
          puts("_uheapmin  failed.");
          exit(EXIT_FAILURE);
     }
    return  0;
}
```

"_heapmin — Release Unused Memory from Default Heap" on page 295
"_debug_uheapmin — Release Unused Memory in User Heap" on page 136
"_ucreate — Create a Memory Heap" on page 646
"Differentiating between Memory Management Functions" on page 23
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_uheapset — Validate and Set Memory Heap

Format

#include <umalloc.h>
int _heapset(Heap_t *heap*, unsigned int *fill*);

Description

Language Level: Extension

_uheapset checks the *heap* you specify for minimal consistency by checking all allocated and freed objects on the heap (similar to _uheapchk). It then sets each byte of the heap's free objects to the value of *fill*.

Using _uheapset can help you locate problems where your program continues to use a freed pointer to an object. After you set the free heap to a specific value, when your program tries to interpret the set values in the freed object as data, unexpected results occur, indicating a problem.

_uheapset works just like _heapset, except that you specify the heap to check; _heapset always checks the default heap.

Note: Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap.

Return Value

_uheapset returns one of the following values, defined in both <umalloc.h> and <malloc.h>:

_HEAPBADBEGIN The heap specified is not valid. It may have been closed or destroyed.
_HEAPBADNODE A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY The heap has not been initialized.
_HEAPOK The heap appears to be consistent.

This example creates a heap and allocates and frees memory from it. It then calls _uheapset to set the freed memory to a value.

_uheapset

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
    Heap_t    myheap;
    char      *ptr;
    int       rc;

     /* Use default heap as user heap */
     myheap = _udefault(NULL);

     if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
          puts("Cannot allocate memory from user heap.");
          exit(EXIT_FAILURE);
     }
     memset(ptr, 'A', 10);
     free(ptr);

     if (_HEAPOK != (rc = _uheapset(myheap, 'x'))) {
          switch(rc)  {
               case  _HEAPEMPTY:
                     puts("The heap has not been initialized.");
                     break;
               case  _HEAPBADNODE:
                     puts("A memory node is corrupted or the heap is damaged.");
                     break;
               case  _HEAPBADBEGIN:
                     puts("The heap specified is not valid.");
                     break;
          }
          exit(rc);
     }
     return  0;
}
```

"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heapset — Validate and Set Default Heap" on page 296
"_uheapchk — Validate Memory Heap" on page 665
"_uheap_walk — Return Information about Memory Heap" on page 671
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_uheap_walk — Return Information about Memory Heap

```
#include <umalloc.h>
int _uheap_walk(Heap_t heap, int (*callback_fn)(const void *object,
                        size_t size, int flag, int status,
                        const char* file, int line) );
```

Description

Language Level: Extension

_uheap_walk traverses the *heap* you specify, and, for each allocated or freed object, it calls the *callback_fn* function that you provide. _uheap_walk works just like _heap_walk, except that you specify the heap to be traversed; _heap_walk always traverses the default heap.

For each object, _uheap_walk passes your function:

object A pointer to the object.

size The size of the object.

flag The value _USEDENTRY, if the object is currently allocated, or _FREEENTRY, if the object has been freed. (Both values are defined in <malloc.h>.)

status One of the following values, defined in both <umalloc.h> and <malloc.h>, depending on the status of the object:

- **_HEAPBADBEGIN** The heap specified is not valid. It may have been closed or destroyed.
- **_HEAPBADNODE** A memory node is corrupted, or the heap is damaged.
- **_HEAPEMPTY** The heap has not been initialized.
- **_HEAPOK** The heap appears to be consistent.

file The name of the file where the object was allocated or freed.

line The line where the object was allocated or freed.

_uheap_walk provides information about all objects, regardless of which memory management functions were used to allocate them. However, the *file* and *line* information are only available if the object was allocated and freed using the debug versions of the memory management functions. Otherwise, *file* is NULL and *line* is 0.

_uheap_walk calls *callback_fn* for each object until one of the following occurs:

All objects have been traversed.
callback_fn returns a nonzero value to _heap_walk.
It cannot continue because of a problem with the heap.

You may want to code your *callback_fn* to return a nonzero value if the status of the object is not _HEAPOK. Even if *callback_fn* returns 0 for an object that is

corrupted, _heap_walk cannot continue because of the state of the heap and returns to its caller.

You can use *callback_fn* to process information from _uheap_walk in various ways. For example, you may want to print the information to a file, or use it to generate your own error messages. You can use the information to look for memory leaks and objects incorrectly allocated or freed from the heap. It can be especially useful to call _uheap_walk when _uheapchk returns an error.

Notes:

1. Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap
2. _uheap_walk locks the heap while it traverses it, to ensure that no other operations use the heap until _uheap_walk finishes. As a result, in your *callback_fn*, you cannot call any critical functions in the runtime library, either explicitly or by calling another function that calls a critical function. See the *Programming Guide* for a list of critical functions.

Return Value

_uheap_walk returns the last value of *status* to the caller.

This example creates a heap and performs memory operations on it. _uheap_walk then traverses the heap and calls callback_function for each memory object. The callback_function prints a message about each memory block.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int _Optlink callback_function(const void *pentry, size_t sz, int useflag,
                                        int status, const char *filename, size_t line)
{
    if (_HEAPOK != status) {
        puts("status is not _HEAPOK.");
       exit(status);
   }
    if (_USEDENTRY == useflag)
        printf("allocated   %p        %u\n", pentry, sz);
   else
        printf("freed         %p        %u\n",  pentry,  sz);
   return  0;
}

int main(void)
{
   Heap_t   myheap;
    char       *p1, *p2, *p3;

    /* User default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (p1 = (char*)_umalloc(myheap, 100)) ||
          NULL == (p2 = (char*)_umalloc(myheap, 200)) ||
          NULL == (p3 = (char*)_umalloc(myheap, 300))) {
        puts("Cannot allocate memory from user heap.");
       exit(EXIT_FAILURE);
   }
   free(p2);
   puts("usage          address    size\n-----          -------    ----");

   _uheap_walk(myheap,  callback_function);

   free(p1);
   free(p3);
   return  0;

   /*************************************************************************
        The output should be similar to :

       usage          address     size
       -----          -------     ----
       allocated   73A20         300
       allocated   738C0         100
         :
         :
       freed         73930         224
   *************************************************************************/
}
```

"_heapmin — Release Unused Memory from Default Heap" on page 295
"_heap_walk — Return Information about Default Heap" on page 298

_uheap_walk

"_uheapchk — Validate Memory Heap" on page 665
"_uheapset — Validate and Set Memory Heap" on page 669
"Managing Memory" in the *Programming Guide*
"Debugging Your Heaps" in the *Programming Guide*
"<umalloc.h>" on page 779

_ultoa — Convert Unsigned Long Integer to String

Format

```
#include <stdlib.h>
char *_ultoa(unsigned long value, char *string, int radix);
```

Description

Language Level: Extension

_ultoa converts the digits of the given unsigned long *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range of 2 through 36.

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes, including the null character (\0).

Return Value

_ultoa returns a pointer to *string*. There is no error return value.

This example converts the digits of the value 255 to decimal, binary, and hexadecimal representations.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  buffer[10];
   char  *p;

    p = _ultoa(255UL, buffer, 10);
    printf("The result of _ultoa(255) with radix of 10 is %s\n", p);
    p = _ultoa(255UL, buffer, 2);
    printf("The result of _ultoa(255) with radix of 2 is %s\n", p);
    p = _ultoa(255UL, buffer, 16);
    printf("The result of _ultoa(255) with radix of 16 is %s\n", p);
   return  0;

   /*****************************************************************************
        The output should be:

        The result of _ultoa(255) with radix of 10 is 255
        The result of _ultoa(255) with radix of 2 is 11111111
        The result of _ultoa(255) with radix of 16 is ff
   *****************************************************************************/
}
```

"_ecvt — Convert Floating-Point to Character" on page 166
"_fcvt — Convert Floating-Point to String" on page 192
"_gcvt — Convert Floating-Point to String" on page 268
"_itoa — Convert Integer to String" on page 336
"_ltoa — Convert Long Integer to String" on page 367
"<stdlib.h>" on page 775

_ulltoa — Convert Unsigned Long Long Integer to String

Format

```
#include <stdlib.h>
char *_ulltoa(unsigned long long value, char *string, int radix);
```

Description

Language Level: Extension

_ulltoa converts the digits of the given unsigned long long *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range of 2 through 36.

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 65 bytes, including the null character (\0).

Return Value

_ulltoa returns a pointer to *string*. There is no error return value.

This example converts the digits of the value 255 to decimal, binary, and hexadecimal representations.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char  buffer[10];
   char  *p;

    p = _ulltoa(255ULL, buffer, 10);
    printf("The result of _ulltoa(255) with radix of 10 is %s\n", p);
    p = _ulltoa(255ULL, buffer, 2);
    printf("The result of _ulltoa(255) with radix of 2 is %s\n", p);
    _ulltoa(255ULL, buffer, 16);
    printf("The result of _ulltoa(255) with radix of 16 is %s\n", buffer);
   return  0;

   /*****************************************************************************
        The output should be:

        The result of _ulltoa(255) with radix of 10 is 255
        The result of _ulltoa(255) with radix of 2 is 11111111
        The result of _ulltoa(255) with radix of 16 is ff
   *****************************************************************************/
}
```

"_ecvt — Convert Floating-Point to Character" on page 166
"_fcvt — Convert Floating-Point to String" on page 192
"_gcvt — Convert Floating-Point to String" on page 268
"_itoa — Convert Integer to String" on page 336
"_ltoa — Convert Long Integer to String" on page 367
"_ultoa — Convert Unsigned Long Integer to String" on page 675
"<stdlib.h>" on page 775

_umalloc — Reserve Memory Blocks from User Heap

Format

```
#include <umalloc.h>
void *_umalloc(Heap_t heap, size_t size);
```

Description

Language Level: Extension

_umalloc allocates a memory block of *size* bytes from the *heap* you specify. Unlike _ucalloc, _umalloc does not initialize all bits to 0.

_umalloc works just like malloc, except that you specify the heap to use; malloc always allocates from the default heap. A debug version of this function, _debug_umalloc, is also provided.

If the *heap* does not have enough memory for the request, _umalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _umalloc, use the non-heap-specific realloc and free. These functions always check what heap the memory was allocated from.

Return Value

_umalloc returns a pointer to the reserved space. If *size* was specified as 0, or if your *getmore_fn* cannot provide enough memory, _umalloc returns NULL. Passing _umalloc a heap that is not valid results in undefined behavior.

This example creates a heap and uses _umalloc to allocate memory from the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t   myheap;
   char     *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
        puts("Cannot allocate memory from user heap.");
        exit(EXIT_FAILURE);
   }
   free(ptr);
   return  0;
}
```

"calloc — Reserve and Initialize Storage" on page 75
"_debug_umalloc — Reserve Memory Blocks from User Heap" on page 138
"free — Release Storage Blocks" on page 238

_umalloc

"malloc — Reserve Storage Block" on page 376
"realloc — Change Reserved Storage Block Size" on page 452
"_ucalloc — Reserve and Initialize Memory from User Heap" on page 641
"_ucreate — Create a Memory Heap" on page 646
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

umask — Sets File Mask of Current Process

Format

```
#include <io.h>
#include <sys\stat.h>
int umask(int pmode);
```

Description

Language Level: XPG4, Extension

umask sets the file permission mask of the environment for the currently running process to the mode specified by *pmode*. The file permission mask modifies the permission setting of new files created by creat, open, or _sopen.

If a bit in the mask is 1, the corresponding bit in the requested permission value of the file is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The possible values for *pmode* are defined in <sys\stat.h>:

Value	Meaning
S_IREAD	No effect
S_IWRITE	Writing not permitted
S_IREAD | S_IWRITE	Writing not permitted.

If the write bit is set in the mask, any new files will be read-only. You cannot give write-only permission, meaning that setting the read bit has no effect.

Note: In earlier releases of VisualAge C++, umask began with an underscore (_umask). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _umask to umask for you.

Return Value

umask returns the previous value of *pmode*. A return value of –1 indicates that the value used for *pmode* was not valid, and errno is set to EINVAL.

This example sets the permission mask to create a write-only file.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

int main(void)
{
   int  oldMask;

    oldMask = umask(S_IWRITE);
    printf("\nDefault system startup mask is %d.\n", oldMask);
   return  0;

   /*************************************************************************
        The output should be:

        Default system startup mask is 0.
   *************************************************************************/
}
```

"chmod — Change File Permission Setting" on page 83
"creat — Create New File" on page 100
"open — Open File" on page 418
"_sopen — Open Shared File" on page 516
"<io.h>" on page 764
"<sys\stat.h>" on page 778

ungetc — Push Character onto Input Stream

Format

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

Language Level: ANSI, SAA, POSIX, XPG4

ungetc pushes the unsigned character *c* back onto the given input *stream*. However, only one sequential character is guaranteed to be pushed back onto the input stream if you call ungetc consecutively. The *stream* must be open for reading. A subsequent read operation on the *stream* starts with *c*. The character *c* cannot be the EOF character.

Characters placed on the stream by ungetc will be erased if fseek, fsetpos, rewind, or fflush is called before the character is read from the *stream*.

Return Value

ungetc returns the integer argument *c* converted to an unsigned char, or EOF if *c* cannot be pushed back.

In this example, the while statement reads decimal digits from an input data stream by using arithmetic statements to compose the numeric values of the numbers as it reads them. When a nondigit character appears before the end of the file, ungetc replaces it in the input stream so that later input functions can process it.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int  ch;
     unsigned int result = 0;

     while (EOF != (ch = getc(stdin)) && isdigit(ch))
          result = result * 10 + ch -'0';
     if (EOF != ch)
          /* Push back the nondigit character onto the input stream                    */
          ungetc(ch,  stdin);

     printf("Input number : %d\n", result);
    return  0;

   /*************************************************************************
          For the following input:

          12345s

          The output should be:

          Input number : 12345
    *************************************************************************/
}
```

ungetc

"getc – getchar — Read a Character" on page 270
"fflush — Write Buffer to File" on page 199
"fseek — Reposition File Position" on page 247
"fsetpos — Set File Position" on page 249
"putc – putchar — Write a Character" on page 436
"rewind — Adjust Current File Position" on page 465
"_ungetch — Push Character Back to Keyboard" on page 683
"<stdio.h>" on page 774

_ungetch — Push Character Back to Keyboard

Format

```
#include <conio.h>
int _ungetch(int c);
```

Description

Language Level: Extension

_ungetch pushes the character *c* back to the keyboard, causing *c* to be the next character read. _ungetch fails if called more than once before the next read operation. The character *c* cannot be the EOF character.

Return Value

If successful, _ungetch returns the character *c*. A return value of EOF indicates an error.

This example uses _getch to read a string delimited by the character 'x'. It then calls _ungetch to return the delimiter to the keyboard buffer. Other input routines can then process the delimiter.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
    int  ch;
     printf("\nType in some letters.\n");
     printf("If you type in an 'x', the program ends.\n");
    for(;;)  {
          ch = _getch();
          if ('x' == ch) {
               _ungetch(ch);
               break;
          }
          _putch(ch);
    }
     ch = _getch();
    printf("\n");
     printf("\nThe last character was '%c'.", ch);
    return  0;
```

```
    /*************************************************************
        Here is the output from a sample run:

        Type in some letters.
        If you type in an 'x', the program ends.
        One Two Three Four Five Si
        The last character was 'x'.
    *************************************************************/
}
```

"_cscanf — Read Data from Keyboard" on page 112
"_getch - _getche — Read Character from Keyboard" on page 272
"_putch — Write Character to Screen" on page 438
"ungetc — Push Character onto Input Stream" on page 681
"<conio.h>" on page 762

ungetwc — Push Wide Character onto Input Stream

Format

```
#include <wchar.h>
wint_t ungetwc(wint_t wc, FILE *stream);
```

Description

Language Level: XPG4

ungetwc pushes the wide character by *wc* back onto the input *stream*. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (on the *stream*) to a file positioning function (fseek, fsetpos, or rewind) discards any pushed-back wide characters for the stream.

The external storage corresponding to the stream is unchanged. There is always at least one wide character of push-back.

If the value of *wc* is WEOF, the operation fails and the input stream is unchanged.

A successful call to the ungetwc function clears the EOF indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.

For a text stream, the file position indicator is backed up by one wide character. This affects ftell, fflush, fseek (with SEEK_CUR), and fgetpos.

For a binary stream, the position indicator is unspecified until all characters are read or discarded, unless the last character is pushed back, in which case the file position indicator is backed up by one wide character. This affects ftell, fseek (with SEEK_CUR), fgetpos, and fflush.

After calling ungetwc, flush the buffer or reposition the stream pointer before calling a read function for the stream, unless EOF has been reached. After a read operation on the stream, flush the buffer or reposition the stream pointer before calling ungetwc.

Notes:

1. Under VisualAge for C++, only 1 wide character may be pushed back.
2. The position on the stream after a successful call to ungetwc is one wide character prior to the current position.

Return Value ungetwc returns the wide character pushed back after conversion, or WEOF if the operation fails.

This example reads in wide characters from stream, and then calls ungetwc to push the characters back to the stream.

```
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE               *stream;
    wint_t             wc;
    wint_t             wc2;
     unsigned int result = 0;

     if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
          printf("Unable to open file.\n");
          exit(EXIT_FAILURE);
    }

     while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
          result = result * 10 + wc - L'0';

     if (WEOF != wc)
          ungetwc(wc, stream);       /* Push the nondigit wide character back */

     /* get the pushed back character */
     if (WEOF != (wc2 = fgetwc(stream))) {
          if (wc != wc2) {
               printf("Subsequent fgetwc does not get the pushed back character.\n");
               exit(EXIT_FAILURE);
          }
          printf("The digits read are '%i'\n"
                    "The character being pushed back is '%lc'", result, wc2);
     }
     return  0;

     /*************************************************************************
          Assuming the file ungetwc.dat contains:

          12345ABCDE67890XYZ

          The output should be similar to :

          The digits read are '12345'
          The character being pushed back is 'A'
     *************************************************************************/
}
```

"fflush — Write Buffer to File" on page 199
"fseek — Reposition File Position" on page 247

"fsetpos — Set File Position" on page 249
"getwc — Read Wide Character from Stream" on page 285
"putwc — Write Wide Character" on page 442
"ungetc — Push Character onto Input Stream" on page 681
"_ungetch — Push Character Back to Keyboard" on page 683
"<wchar.h>" on page 780

unlink — Delete File

Format

```
#include <stdio.h>    /* also in <io.h> */
int unlink(const char *pathname);
```

Description

Language Level: XPG4, Extension

unlink deletes the file specified by *pathname*.

Portability Note: For portability, use the ANSI/ISO function remove instead of unlink.

Note: In earlier releases of VisualAge C++, unlink began with an underscore (_unlink). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _unlink to unlink for you.

Return Value

unlink returns 0 if the file is successfully deleted. A return value of -1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCESS	The path name specifies a read-only file or a directory.
EISOPEN	The file is open.
ENOENT	An incorrect path name was specified, or the file or path name was not found.

This example deletes the file tmpfile from the system or prints an error message if unable to delete it.

```
#include <stdio.h>

int main(void)
{
    if (-1 == unlink("tmpfile"))
        perror("Cannot delete tmpfile");
    else
        printf("tmpfile has been successfully deleted\n");
    return  0;

    /*************************************************************************
        If the file "tmpfile" exists, the output should be:

        tmpfile has been successfully deleted
    *************************************************************************/
}
```

"remove — Delete File" on page 463
"_rmtmp — Remove Temporary Files" on page 482
"<stdio.h>" on page 774

_uopen — Open Heap for Use

Format

#include <umalloc.h>
int _uopen(Heap_t *heap*);

Description

Language Level: Extension

_uopen allows the current process to use the *heap* you specify. If the heap is shared, you must call _uopen in each process that will allocate or free from the heap. See "Managing Memory" in the *Programming Guide* for more information about sharing heaps, and about creating and using heaps in general.

Return Value

If successful, _uopen returns 0. A nonzero return code indicates failure. Passing _uopen a heap that is not valid results in undefined behavior.

This example creates a fixed-size heap, then uses _uopen to open it. The program then performs operations on the heap, and closes and destroys it.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   void       *initial_block;
   Heap_t     myheap;
   char       *p;

    /* Call VirtualAlloc to get the initial block of memory */
    if (NULL == (initial_block =
          VirtualAlloc(NULL, 65536, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE)))
   {
        printf("VirtualAlloc error: %d.\n", GetLastError());
       exit(EXIT_FAILURE);
   }

    /* Create a fixed size heap starting with the block declared earlier */
    if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                          _HEAP_REGULAR, NULL, NULL))) {
       puts("_ucreate  failed.");
       exit(EXIT_FAILURE);
   }
```

```
    if (0 != _uopen(myheap)) {
        puts("_uopen  failed.");
        exit(EXIT_FAILURE);
    }
    p = (char *)_umalloc(myheap, 100);
    free(p);
    if (0 != _uclose(myheap)) {
        puts("_uclose  failed");
        exit(EXIT_FAILURE);
    }
    if (FALSE == VirtualFree(initial_block, 0, MEM_RELEASE)) {
         printf("VirtualFree error: %d.\n", GetLastError());
        exit(EXIT_FAILURE);
    }
    return  0;
}
```

"_uaddmem — Add Memory to a Heap" on page 637
"_uclose — Close Heap from Use" on page 643
"_ucreate — Create a Memory Heap" on page 646
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

_ustats — Get Information about Heap

Format

```
#include <umalloc.h>
int _ustats(Heap_t heap, _HEAPSTATS *hpinfo);
```

Description

Language Level: Extension

_ustats gets information about the *heap* you specify and stores it in the *hpinfo* structure you pass to it.

The _HEAPSTATS structure type is defined in <umalloc.h>. The members it contains and the information that _ustats stores in each is as follows:

_provided	How much memory the heap holds (excluding memory used for overhead for the heap)
_used	How much memory is currently allocated from the heap
_shared	Whether the memory is shared (_shared is 1) or not (_shared is 0)
_maxfree	The size of the largest contiguous piece of memory available on the heap

Return Value

If successful, _ustats returns 0. A nonzero return code indicates failure. Passing _ustats a heap that is not valid results in undefined behavior.

This example creates a heap and allocates memory from it. It then calls _ustats to print out information about the heap.

_ustats

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
    Heap_t          myheap;
    _HEAPSTATS    myheap_stat;
    char              *ptr;

     /* Use default heap as user heap */
     myheap = _udefault(NULL);

     if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
          puts("Cannot allocate memory from user heap.");
          exit(EXIT_FAILURE);
     }
     if (0 != _ustats(myheap, &myheap_stat)) {
          puts("_ustats  failed.");
          exit(EXIT_FAILURE);
     }
     printf ("_provided: %u\n", myheap_stat._provided);
     printf ("_used       : %u\n", myheap_stat._used);
     printf ("_tiled      : %u\n", myheap_stat._tiled);
     printf ("_shared    : %u\n", myheap_stat._shared);
     printf ("_max_free: %u\n", myheap_stat._max_free);
    free(ptr);
    return  0;
    /*****************************************************************************
          The output should be similar to :

          _provided:  65264
          _used       :  14304
          _tiled     :  0
          _shared    :  0
          _max_free:  50960
    *****************************************************************************/
}
```

"_mheap — Query Memory Heap for Allocated Object" on page 404
"_ucreate — Create a Memory Heap" on page 646
"Managing Memory" in the *Programming Guide*
"<umalloc.h>" on page 779

utime — Set Modification Time

Format

```
#include <sys\utime.h>
#include <sys\types.h>
int utime(char *pathname, struct utimbuf *times);
```

Description

Language Level: XPG4, Extension

utime sets the modification time for the file specified by *pathname*. The process must have write access to the file; otherwise, the time cannot be changed.

Although the utimbuf structure contains a field for access time, only the modification time is set in the Windows operating system. If *times* is a NULL pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type utimbuf, defined in <sys\utime.h>. The modification time is set from the modtime field in this structure.

utime accepts only even numbers of seconds. If you enter an odd number of seconds, the function rounds it down.

Note: In earlier releases of VisualAge C++, utime began with an underscore (_utime). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _utime to utime for you.

Return Value

utime returns 0 if the file modification time was changed. A return value of -1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCESS	The path name specifies a directory or read-only file.
EMFILE	There are too many open files. You must open the file to change its modification time.
ENOENT	The file path name was not found, or the file name was incorrectly specified.

This example sets the last modification time of file utime.dat to the current time. It prints an error message if it cannot.

```
#include <sys\types.h>
#include <sys\utime.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define   FILENAME          "utime.dat"

int main(void)
{
    struct utimbuf ubuf;
    struct stat statbuf;
    FILE *fp;                                                   /* File pointer         */

    /* creating file, whose date will be changed by calling utime                    */
    fp = fopen(FILENAME, "w");

    /* write Hello World in the file                                                 */
    fprintf(fp, "Hello World\n");

    /* close file                                                                    */
   fclose(fp);

    /* seconds to current date from 1970 Jan 1                                      */
    /* Fri Dec 31 23:59:58 1999                                                      */
    ubuf.modtime = 946702799;

    /* changing file modification time                                               */
    if (-1 == utime(FILENAME, &ubuf)) {
        perror("utime  failed");
        remove(FILENAME);
        return  EXIT_FAILURE;
    }

    /* display the modification time                                                 */
    if (0 == stat(FILENAME, &statbuf))
        printf("The file modification time is %s", ctime(&statbuf.st_mtime));
   else
        printf("File could not be found\n");
   remove(FILENAME);
   return  0;

   /*****************************************************************************
        The output should be:

        The file modification time is Fri Dec 31 23:59:58 1999
   *****************************************************************************/
}
```

"fstat — Information about Open File" on page 255
"stat — Get Information about File or Directory" on page 532
"<sys\utime.h>" on page 778
"<sys\types.h>" on page 778

va_arg - va_end - va_start — Access Function Arguments

Format

#include <stdarg.h>
var_type va_arg(va_list *arg_ptr*, *var_type*);
void va_end(va_list *arg_ptr*);
void va_start(va_list *arg_ptr*, *variable_name*);

Description

Language Level: ANSI, SAA

va_arg, va_end, and va_start access the arguments to a function when it takes a fixed number of required arguments and a variable number of optional arguments. All three of these are macros. You declare required arguments as ordinary parameters to the function and access the arguments through the parameter names.

va_start initializes the *arg_ptr* pointer for subsequent calls to va_arg and va_end.

The argument *variable_name* is the identifier of the rightmost named parameter in the parameter list (preceding , ...). Use va_start before va_arg. Corresponding va_start and va_end macros must be in the same function.

va_arg retrieves a value of the given *var_type* from the location given by *arg_ptr*, and increases *arg_ptr* to point to the next argument in the list. va_arg can retrieve arguments from the list any number of times within the function. The *var_type* argument must be one of int, long, double, struct, union, or pointer, or a typedef of one of these types.

va_end is needed to indicate the end of parameter scanning.

Return Value

va_arg returns the current argument. va_end and va_start do not return a value.

This example passes a variable number of arguments to a function, stores each argument in an array, and prints each argument.

```
#include <stdio.h>
#include <stdarg.h>

int vout(int max,...);

int main(void)
{
     vout(3, "Sat", "Sun", "Mon");
    printf("\n");
     vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
    return  0;
}

int vout(int max,...)
{
    va_list  arg_ptr;
     int args = 0;
    char  *days[7];

    va_start(arg_ptr,  max);
     while (args < max) {
          days[args] = va_arg(arg_ptr, char *);
          printf("Day:   %s   \n",  days[args++]);
    }
    va_end(arg_ptr);

   /*************************************************************************
         The output should be:

        Day:   Sat
        Day:   Sun
        Day:   Mon

        Day:   Mon
        Day:   Tues
        Day:   Wed
        Day:   Thurs
        Day:   Fri
   *************************************************************************/
}
```

"vfprintf — Print Argument Data to Stream" on page 698
"vprintf — Print Argument Data" on page 700
"vsprintf — Print Argument Data to Buffer" on page 702
"<stdarg.h>" on page 773

vfprintf — Print Argument Data to Stream

Format

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

vfprintf formats and writes a series of characters and values to the output *stream*. vfprintf works just like fprintf, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start for each call. In contrast, fprintf can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

vfprintf converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for printf. For a description of the format string, see "printf — Print Formatted Characters" on page 429.

In extended mode, vfprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See "Infinity and NaN Support" on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, vfprintf prints (null). (In previous releases of VisualAge C++, vfprintf produced no output for a null string.)

Return Value

If successful, vfprintf returns the number of bytes written to *stream*. If an error occurs, the function returns a negative value.

This example prints out a variable number of strings to the file vfprintf.out.

```
#include <stdarg.h>
#include <stdio.h>

#define FILENAME "vfprintf.opf"

void vout(FILE *stream, char *fmt,...);

char fmt1[] = "%s    %s    %s    %s\n";

int main(void)
{
   FILE  *stream;

    stream = fopen(FILENAME, "w");
    vout(stream, fmt1, "Sat", "Sun", "Mon", "Tue");
   fclose(stream);
   return  0;

   /*****************************************************************************
        After running the program, the output file should contain:

        Sat   Sun   Mon   Tue
   *****************************************************************************/
}

void vout(FILE *stream, char *fmt,...)
{
   va_list  arg_ptr;

   va_start(arg_ptr,  fmt);
    vfprintf(stream, fmt, arg_ptr);
   va_end(arg_ptr);
}
```

“fprintf — Write Formatted Data to a Stream” on page 224
“printf — Print Formatted Characters” on page 429
“va_arg - va_end - va_start — Access Function Arguments” on page 696
“vprintf — Print Argument Data” on page 700
“vsprintf — Print Argument Data to Buffer” on page 702
“vswprintf — Format and Write Wide Characters to Buffer” on page 704
“Infinity and NaN Support” on page 27
“<stdarg.h>” on page 773
“<stdio.h>” on page 774

vprintf — Print Argument Data

Format

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

vprintf formats and prints a series of characters and values to **stdout**. vprintf works just like printf, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start for each call. In contrast, printf can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

vprintf converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for printf. For a description of the format string, see "printf — Print Formatted Characters" on page 429.

In extended mode, vprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See "Infinity and NaN Support" on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, vfprintf prints (null). (In previous releases of VisualAge C++, vprintf produced no output for a null string.)

Return Value

If successful, vprintf returns the number of bytes written to **stdout**. If an error occurs, vprintf returns a negative value.

This example prints out a variable number of strings to **stdout**.

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);

int main(void)
{
    char fmt1[] = "%s    %s    %s\n";
    vout(fmt1, "Sat", "Sun", "Mon");
   return  0;

   /*****************************************************************************
        The output should be:

        Sat   Sun   Mon
   *****************************************************************************/
}

void vout(char *fmt, ...)
{
   va_list  arg_ptr;

   va_start(arg_ptr,  fmt);
   vprintf(fmt,  arg_ptr);
   va_end(arg_ptr);
}
```

"printf — Print Formatted Characters" on page 429
"va_arg - va_end - va_start — Access Function Arguments" on page 696
"vfprintf — Print Argument Data to Stream" on page 698
"vsprintf — Print Argument Data to Buffer" on page 702
"<stdarg.h>" on page 773
"<stdio.h>" on page 774

vsprintf — Print Argument Data to Buffer

Format

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

Description

Language Level: ANSI, SAA, XPG4, Extension

vsprintf formats and stores a series of characters and values in the buffer *target-string*. vsprintf works just like sprintf, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start for each call. In contrast, sprintf can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

vsprintf converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for printf. For a description of the format string, see "printf — Print Formatted Characters" on page 429.

In extended mode, vsprintf also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers. See "Infinity and NaN Support" on page 27 for more information on infinity and NaN values.

If you specify a null string for the %s or %ls format specifier, vsprintf prints (null). (In previous releases of VisualAge C++, vsprintf produced no output for a null string.)

Return Value

If successful, vsprintf returns the number of bytes written to *target-string*. If an error occurs, vsprintf returns a negative value.

This example assigns a variable number of strings to string and prints the resultant string.

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt,...);

char fmt1[] = "%s    %s    %s\n";

int main(void)
{
    char  string[100];

     vout(string, fmt1, "Sat", "Sun", "Mon");
     printf("The string is: %s", string);
    return  0;

   /*****************************************************************************
        The output should be:

        The string is: Sat    Sun    Mon
   *****************************************************************************/
}

void vout(char *string, char *fmt,...)
{
    va_list  arg_ptr;

    va_start(arg_ptr,  fmt);
     vsprintf(string, fmt, arg_ptr);
    va_end(arg_ptr);
}
```

“printf — Print Formatted Characters” on page 429
“sprintf — Print Formatted Data to Buffer” on page 525
“swprintf — Format and Write Wide Characters to Buffer” on page 609
“va_arg - va_end - va_start — Access Function Arguments” on page 696
“vfprintf — Print Argument Data to Stream” on page 698
“vprintf — Print Argument Data” on page 700
“vswprintf — Format and Write Wide Characters to Buffer” on page 704
“Infinity and NaN Support” on page 27
“<stdarg.h>” on page 773
“<stdio.h>” on page 774

vswprintf — Format and Write Wide Characters to Buffer

Format

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t *wcsbuffer, size_t n,
              const wchar_t *format, va_list argptr);
```

Description

Language Level: ANSI 93

vswprintf formats and stores a series of wide characters and values in the buffer *wcsbuffer*. vswprintf works just like swprintf, except that *argptr* points to a list of wide-character arguments whose number can vary from call to call. These arguments should be initialized by va_start for each call. In contrast, swprintf can have a list of arguments, but the number of arguments in that list are fixed when you compile in the program.

The value *n* specifies the maximum number of wide characters to be written, including the terminating null character.

vswprintf converts each entry in the argument list according to the corresponding wide-character format specifier in *format*. The *format* has the same form and function as the format string for printf, with the following exceptions:

- %c (without an l prefix) converts an integer argument to **wchar_t**, as if by calling mbtowc.
- %lc converts a **wint_t** to **wchar_t**.
- %s (without an l prefix); converts an array of multibyte characters to an array of **wchar_t**, as if by calling mbrtowc. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.
- %ls writes an array of **wchar_t**. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.

For a complete description of format specifiers, see “printf — Print Formatted Characters” on page 429.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

If you specify a null string for the %s or %ls format specifier, vswprintf prints (null).

Return Value vswprintf returns the number of bytes written in the array, not counting the terminating null wide character.

This example creates a function vout that takes a variable number of wide-character arguments and uses vswprintf to print them to wcstr.

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

wchar_t *format3 = L"%ls   %d   %ls";
wchar_t *format5 = L"%ls   %d   %ls   %d   %ls";

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...)
{
   va_list  arg_ptr;

   va_start(arg_ptr,  fmt);
    vswprintf(wcs, n, fmt, arg_ptr);
   va_end(arg_ptr);
   return;
}

int main(void)
{
   wchar_t  wcstr[100];

    vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
   printf("%ls\n",  wcstr);
    vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
   printf("%ls\n",  wcstr);
   return  0;

   /*************************************************************************
        The output should be similar to :

        ONE   2   THREE
        ONE   2   THREE   4   FIVE
   *************************************************************************/
}
```

"printf — Print Formatted Characters" on page 429
"swprintf — Format and Write Wide Characters to Buffer" on page 609
"vfprintf — Print Argument Data to Stream" on page 698
"vprintf — Print Argument Data" on page 700
"vsprintf — Print Argument Data to Buffer" on page 702
"<stdarg.h>" on page 773
"<wchar.h>" on page 780

wcrtomb — Convert Wide Character to Multibyte Character

Format

```
#include <wchar.h>
int wcrtomb(char *dest, wchar_t wc, mbstate_t *ps);
```

Description

Language Level: ANSI 93

wcrtomb is a restartable version of wctomb, and performs the same function. It first determines the length of the wide character pointed to by *wc*. It then converts the wide character to the corresponding multibyte character, and stores the converted character in the location pointed to by *dest*, if *dest* is not a null pointer. A maximum of MB_CUR_MAX bytes are stored.

With wcrtomb, you can switch from one multibyte string to another. On systems that support shift states, *ps* represents the initial shift state of the string (0). If you read in only part of the string, wcrtomb sets *ps* to the string's shift state at the point you stopped. You can then call wcrtomb again for that string and pass in the updated *ps* value to continue reading where you left off.

The behavior of wcrtomb is affected by the LC_CTYPE category of the current locale.

Return Value

If *dest* is a null pointer, wcrtomb returns 0. If *dest* is not a null pointer, wcrtomb returns the number of bytes stored in *dest*. If *wc* is not a valid wide character, wcrtomb sets errno to EILSEQ and returns −1.

This example uses wcrtomb to convert two wide-character strings to multibyte strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

#define SIZE    20
#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    wchar_t       wcs1[] = L"abc";
    wchar_t       wcs2[] = L"A\x8142" L"C";
    mbstate_t   ss1 = 0;
    mbstate_t   ss2 = 0;
    size_t        length1 = 0;
    size_t        length2 = 0;
   char           mb1[SIZE],  mb2[SIZE];
   int            i;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
    }
    for (i = 0; i <=2; ++i) {
        length1 += wcrtomb(mb1 + length1, *(wcs1 + i), &ss1);
        length2 += wcrtomb(mb2 + length2, *(wcs2 + i), &ss2);
    }
    mb1[length1] = 0;
    mb2[length2] = 0;
    printf("The first multibyte string is: <%s>\n", mb1);
    printf("The second multibyte string is: <%s>\n", mb2);
   return  0;

   /*************************************************************************
        The output should be similiar to :

        The first multibyte string is: <abc>
        The second multibyte string is: <A BC>
   *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbrlen — Calculate Length of Multibyte Character" on page 384
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbsrtowcs — Convert Multibyte String to Wide-Character String" on page 389
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"wctomb — Convert Wide Character to Multibyte Character" on page 753
"<wchar.h>" on page 780

wcscat — Concatenate Wide-Character Strings

Format

#include <wcstr.h>
wchar_t *wcscat(wchar_t *_string1_, const wchar_t *_string2_);

Description

Language Level: SAA, XPG4

wcscat appends a copy of the string pointed to by _string2_ to the end of the string pointed to by _string1_.

wcscat operates on null-terminated wchar_t strings. The string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed.

Return Value

wcscat returns a pointer to the concatenated _string1_.

This example creates the wide character string "computer program" using wcscat.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
    wchar_t *string = L" program";
   wchar_t  *ptr;

    ptr = wcscat(buffer1, string);
    printf("buffer1 = %ls\n", buffer1);
   return  0;

   /*****************************************************************************
        The output should be:

        buffer1 = computer program
   *****************************************************************************/
}
```

"strcat — Concatenate Strings" on page 536
"strncat — Concatenate Strings" on page 566
"wcsncat — Concatenate Wide-Character Strings" on page 723
"<wcstr.h>" on page 782

wcschr — Search for Wide Character

Format

```
#include <wcstr.h>
wchar_t *wcschr(const wchar_t *string, wchar_t character);
```

Description

Language Level: SAA, XPG4

wcschr searches the wide-character *string* for the occurrence of *character*. The *character* can be a wchar_t null character (\0); the wchar_t null character at the end of *string* is included in the search.

wcschr operates on null-terminated wchar_t strings. The string argument to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcschr returns a pointer to the first occurrence of *character* in *string*. If the character is not found, a NULL pointer is returned.

This example finds the first occurrence of the character p in the wide-character string "computer program".

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
   wchar_t  *ptr;
    wchar_t ch = L'p';

    ptr = wcschr(buffer1, ch);
    printf("The first occurrence of %lc in '%ls' is '%ls'\n", ch, buffer1, ptr);
   return  0;

  /*****************************************************************************
       The output should be:

       The first occurrence of p in 'computer program' is 'puter program'
  *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcspbrk — Locate Wide Characters in String" on page 729
"wcsrchr — Locate Wide Character in String" on page 731

wcschr

"wcsspn — Search Wide-Character Strings" on page 735
"wcswcs — Locate Wide-Character Substring" on page 747
"<wcstr.h>" on page 782

wcscmp — Compare Wide-Character Strings

Format

```
#include <wcstr.h>
int wcscmp(const wchar_t *string1, const wchar_t *string2);
```

Description

Language Level: SAA, XPG4

wcscmp compares two wide-character strings.

wcscmp operates on null-terminated wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcscmp returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* identical to *string2*
Greater than 0	*string1* greater than *string2*.

This example compares the wide-character string string1 to string2 using wcscmp.

```
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
   int  result;
    wchar_t string1[] = L"abcdef";
    wchar_t string2[] = L"abcdefg";

    result = wcscmp(string1, string2);
    if (0 == result)
        printf("\"%ls\" is identical to \"%ls\"\n", string1, string2);
   else
        if (result < 0)
            printf("\"%ls\" is less than \"%ls\"\n", string1, string2);
        else
            printf("\"%ls\" is greater than \"%ls\"\n", string1, string2);
   return  0;

   /*************************************************************************
        The output should be:

        "abcdef" is less than "abcdefg"
   *************************************************************************/
}
```

"strcmp — Compare Strings" on page 538
"strcmpi — Compare Strings Without Case Sensitivity" on page 540
"stricmp — Compare Strings as Lowercase" on page 562
"strnicmp — Compare Strings Without Case Sensitivity" on page 572

wcscmp

"wcsncmp — Compare Wide-Character Strings" on page 725
"<wcstr.h>" on page 782

wcscoll — Compare Wide-Character Strings

Format

```
#include <wchar.h>
int wcscoll(const wchar_t *wcstr1, const wchar_t *wcstr2);
```

Description

Language Level: ANSI 93, XPG4

wcscoll compares the wide-character string pointed to by *wcstr1* to the wide-character string pointed to by *wcstr2*, both interpreted according to the information in the LC_COLLATE category of the current locale.

wcscoll differs from the wcscmp function. wcscoll performs a comparison between two wide-character strings based on language collation rules as controlled by the LC_COLLATE category. In contrast, wcscmp performs a wide-character code to wide-character code comparison.

Return Value

wcscoll returns an integer value indicating the relationship between the strings, as listed below:

Value	Meaning
Less than 0	*wcstr1* less than *wcstr2*
0	*wcstr1* equivalent to *wcstr2*
Greater than 0	*wcstr1* greater than *wcstr2*

If *wcs1* or *wcs2* contain characters outside the domain of the collating sequence, wcscoll sets **errno** to EILSEQ. If an error occurs, wcscoll sets **errno** to a nonzero value. There is no error return value.

This example uses wcscoll to compare two wide-character strings.

wcscoll

```
#include <wchar.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    wchar_t *wcs1 = L"A wide string";
    wchar_t *wcs2 = L"a wide string";
   int         result;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
   }
    result = wcscoll(wcs1, wcs2);
    if (0 == result)
        printf("\"%ls\" is identical to \"%ls\"\n", wcs1, wcs2);
    else if (0 > result)
        printf("\"%ls\" is less than \"%ls\"\n", wcs1, wcs2);
   else
        printf("\"%ls\" is greater than \"%ls\"\n", wcs1, wcs2);
   return  0;

   /*****************************************************************************
        The output should be similar to :

        "A wide string" is less than "a wide string"
   *****************************************************************************/
}
```

"strcoll — Compare Strings Using Collation Rules" on page 542
"setlocale — Set Locale" on page 499
"wcscmp — Compare Wide-Character Strings" on page 711
"<wchar.h>" on page 780

wcscpy — Copy Wide-Character Strings

Format

```
#include <wcstr.h>
wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);
```

Description

Language Level: SAA, XPG4

wcscpy copies the contents of *string2* (including the ending wchar_t null character) into *string1*.

wcscpy operates on null-terminated wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed.

Return Value

wcscpy returns a pointer to *string1*.

This example copies the contents of source to destination.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
     wchar_t source[SIZE] = L"This is the source string";
     wchar_t destination[SIZE] = L"And this is the destination string";
    wchar_t  *return_string;

     printf("destination is originally = \"%ls\"\n", destination);
     return_string = wcscpy(destination, source);
     printf("After wcscpy, destination becomes \"%ls\"\n", return_string);
    return  0;

   /*****************************************************************************
        The output should be:

        destination is originally = "And this is the destination string"
        After wcscpy, destination becomes "This is the source string"
   *****************************************************************************/
}
```

wcscpy

"strcpy — Copy Strings" on page 544
"strdup — Duplicate String" on page 549
"strncpy — Copy Strings" on page 570
"wcsncpy — Copy Wide-Character Strings" on page 727
"<wcstr.h>" on page 782

wcscspn — Find Offset of First Wide-Character Match

Format

```
#include <wcstr.h>
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

Description

Language Level: SAA, XPG4

wcscspn determines the number of wchar_t characters in the initial segment of the string pointed to by *string1* that do not appear in the string pointed to by *string2*.

wcscspn operates on null-terminated wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcscspn returns the number of wchar_t characters in the segment.

This example uses wcscspn to find the first occurrence of any of the characters a, x, l, or e in string.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
    wchar_t string[SIZE] = L"This is the source string";
    wchar_t *substring = L"axle";

    printf("The first %i characters in the string \"%ls\" are not in the "
        "string \"%ls\" \n", wcscspn(string, substring), string, substring);
   return  0;

   /*************************************************************************
        The output should be:

        The first 10 characters in the string "This is the source string" are
        not in the string "axle"
   *************************************************************************/
}
```

wcscspn

"strcspn — Compare Strings for Substrings" on page 546
"strspn — Search Strings" on page 585
"wcsspn — Search Wide-Character Strings" on page 735
"wcswcs — Locate Wide-Character Substring" on page 747
"<wcstr.h>" on page 782

wcsftime — Convert to Formatted Date and Time

Format

```
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
                const wchar_t *format, const struct tm *timeptr);
```

Description

Language Level: ANSI 93, XPG4

wcsftime converts the time and date specification in the *timeptr* structure into a wide-character string. It then stores the null-terminated string in the array pointed to by *wdest* according to the format string pointed to by *format*. *maxsize* specifies the maximum number of wide characters that can be copied into the array.

wcsftime works just like strftime, except that it uses wide characters.

The format string is a multibyte character string containing:

Conversion specification characters.
Ordinary wide characters, which are copied into the array unchanged.

The characters that are converted are determined by the LC_TIME category of the current locale and by the values in the time structure pointed to by *timeptr*. The time structure pointed to by *timeptr* is usually obtained by calling the gmtime or localtime function.

For details on the conversion specifiers you can use in the format string, see "strftime — Convert to Formatted Time" on page 557.

Return Value

If the total number of wide characters in the resulting string, including the terminating null wide character, does not exceed *maxsize*, wcsftime returns the number of wide characters placed into *wdest*, not including the terminating null wide character. Otherwise, wcsftime returns 0 and the contents of the array are indeterminate.

This example obtains the date and time using localtime, formats the information with wcsftime, and prints the date and time.

```
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
     struct tm *timeptr;
    wchar_t     dest[100];
    time_t       temp;
    size_t       rc;

     temp = time(NULL);
     timeptr = localtime(&temp);
     rc = wcsftime(dest, sizeof(dest)-1, L" Today is %A,"
                          L" %b %d.\n Time: %I:%M %p", timeptr);
     printf("%d characters placed in string to make:\n\n%ls\n", rc, dest);
    return  0;

    /*****************************************************************************
         The output should be similar to :

         43 characters placed in string to make:

          Today is Thursday, Nov 10.
          Time: 04:56 PM
    *****************************************************************************/
}
```

"gmtime — Convert Time" on page 289
"localtime — Convert Time" on page 356
"strftime — Convert to Formatted Time" on page 557
"strptime — Convert to Formatted Date and Time" on page 576
"<wchar.h>" on page 780

wcsid — Determine Character Set ID for Wide Character

Format

```
#include <stdlib.h>
int wcsid(const wchar_t c);
```

Description

Language Level: Extension

wcsid determines the character set identifier for the specified wide character *wc*. You can specify character set identifiers for each character in the charmap file for a locale. For more information about character set identifiers and charmap files, see the section on internationalization in the *Programming Guide*.

Return Value

wcsid returns the character set identifier for the wide character, or −1 if the wide character is not valid.

This example uses wcsid to get the character set identifier for the wide character A.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    wchar_t wc = L'A';
   int         rc;

    rc = wcsid(wc);
    printf("wide character '%c' is in character set id %i\n", wc, rc);
   return  0;

   /*************************************************************************
        The output should be similar to :

        wide character 'A' is in character set id 0
   *************************************************************************/
}
```

"<stdlib.h>" on page 775
"csid — Determine Character Set ID for Multibyte Character" on page 111

wcslen — Calculate Length of Wide-Character String

Format

```
#include <wcstr.h>
size_t wcslen(const wchar_t *string);
```

Description

Language Level: SAA, XPG4

wcslen computes the number of wide characters in the string pointed to by *string*.

Return Value

wcslen returns the number of wide characters in *string*, excluding the terminating wchar_t null character.

This example computes the length of the wide-character string string.

```
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t *string = L"abcdef";

    printf("Length of \"%ls\" is %i\n", string, wcslen(string));
   return  0;

   /*************************************************************************
       The output should be:

       Length of "abcdef" is 6
   *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"strlen — Determine String Length" on page 564
"<wcstr.h>" on page 782

wcsncat — Concatenate Wide-Character Strings

Format

```
#include <wcstr.h>
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

Description

Language Level: SAA, XPG4

wcsncat appends up to *count* wide characters from *string2* to the end of *string1*, and appends a wchar_t null character to the result.

wcsncat operates on null-terminated wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcsncat returns *string1*.

This example demonstrates the difference between wcscat and wcsncat. wcscat appends the entire second string to the first; wcsncat appends only the specified number of characters in the second string to the first.

```
#include <stdio.h>
#include <wcstr.h>
#include <string.h>

#define   SIZE              40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
   wchar_t  *ptr;

    /* Call wcscat with buffer1 and " program"                                  */

    ptr = wcscat(buffer1, L" program");
    printf("wcscat : buffer1 = \"%ls\"\n", buffer1);

    /* Reset buffer1 to contain just the string "computer" again                */

    memset(buffer1, L'\0', sizeof(buffer1));
    ptr = wcscpy(buffer1, L"computer");

    /* Call wcsncat with buffer1 and " program"                                 */

    ptr = wcsncat(buffer1, L" program", 3);
    printf("wcsncat: buffer1 = \"%ls\"\n", buffer1);
   return  0;

  /*****************************************************************************
       The output should be:

       wcscat : buffer1 = "computer program"
       wcsncat: buffer1 = "computer pr"
  *****************************************************************************/
}
```

wcsncat

"strncat — Concatenate Strings" on page 566
"strcat — Concatenate Strings" on page 536
"wcscat — Concatenate Wide-Character Strings" on page 708
"wcsncmp — Compare Wide-Character Strings" on page 725
"wcsncpy — Copy Wide-Character Strings" on page 727
"<wcstr.h>" on page 782

wcsncmp — Compare Wide-Character Strings

Format

#include <wcstr.h>
int wcsncmp(const wchar_t *_string1_, const wchar_t *_string2_, size_t _count_);

Description

Language Level: SAA, XPG4

wcsncmp compares up to _count_ wide characters in _string1_ to _string2_.

wcsncmp operates on null-terminated wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcsncmp returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	_string1_ less than _string2_
0	_string1_ identical to _string2_
Greater than 0	_string1_ greater than _string2_.

This example demonstrates the difference between wcscmp, which compares the entire strings, and wcsncmp, which compares only a specified number of wide characters in the strings.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              10

int main(void)
{
    int  result;
     int index = 3;
     wchar_t buffer1[SIZE] = L"abcdefg";
     wchar_t buffer2[SIZE] = L"abcfg";
     void print_result(int, wchar_t *, wchar_t *);
```

```
    result = wcscmp(buffer1, buffer2);
    printf("Comparison of each character\n");
   printf("    wcscmp:  ");
    print_result(result, buffer1, buffer2);
    result = wcsncmp(buffer1, buffer2, index);
    printf("\nComparison of only the first %i characters\n", index);
   printf("    wcsncmp:  ");
    print_result(result, buffer1, buffer2);
   return  0;

   /*****************************************************************************
        The output should be:

        Comparison of each character
           wcscmp: "abcdefg" is less than "abcfg"

        Comparison of only the first 3 characters
           wcsncmp: "abcdefg" is identical to "abcfg"
   *****************************************************************************/
}

void print_result(int res,wchar_t *p_buffer1,wchar_t *p_buffer2)
{
    if (0 == res)
        printf("\"%ls\" is identical to \"%ls\"\n", p_buffer1, p_buffer2);
   else
        if (res < 0)
            printf("\"%ls\" is less than \"%ls\"\n", p_buffer1, p_buffer2);
        else
            printf("\"%ls\" is greater than \"%ls\"\n", p_buffer1, p_buffer2);
}
```


“strncmp — Compare Strings” on page 568
“strcmp — Compare Strings” on page 538
“strcoll — Compare Strings Using Collation Rules” on page 542
“strcmpi — Compare Strings Without Case Sensitivity” on page 540
“stricmp — Compare Strings as Lowercase” on page 562
“strnicmp — Compare Strings Without Case Sensitivity” on page 572
“wcscmp — Compare Wide-Character Strings” on page 711
“wcsncat — Concatenate Wide-Character Strings” on page 723
“wcsncpy — Copy Wide-Character Strings” on page 727
“<wcstr.h>” on page 782

wcsncpy — Copy Wide-Character Strings

Format

#include <wcstr.h>
wchar_t *wcsncpy(wchar_t *_string1_, const wchar_t *_string2_, size_t _count_);

Description

Language Level: SAA, XPG4

wcsncpy copies up to _count_ wide characters from _string2_ to _string1_. If _string2_ is shorter than _count_ characters, _string1_ is padded out to _count_ characters with wchar_t null characters.

wcsncpy operates on null-terminated wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

Return Value

wcsncpy returns a pointer to _string1_.

This example demonstrates the difference between wcscpy, which copies the entire wide-character string, and wcsncpy, which copies a specified number of wide characters from the string.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
     wchar_t source[SIZE] = L"123456789";
     wchar_t source1[SIZE] = L"123456789";
     wchar_t destination[SIZE] = L"abcdefg";
     wchar_t destination1[SIZE] = L"abcdefg";
    wchar_t  *return_string;
     int index = 5;
```

```
    /* This is how wcscpy works                                                */

    printf("destination is originally = '%ls'\n", destination);
    return_string = wcscpy(destination, source);
    printf("After wcscpy, destination becomes '%ls'\n\n", return_string);

    /* This is how wcsncpy works                                               */

    printf("destination1 is originally = '%ls'\n", destination1);
    return_string = wcsncpy(destination1, source1, index);
    printf("After wcsncpy, destination1 becomes '%ls'\n", return_string);
   return  0;

  /*****************************************************************************
       The output should be:

       destination is originally = 'abcdefg'
       After wcscpy, destination becomes '123456789'

       destination1 is originally = 'abcdefg'
       After wcsncpy, destination1 becomes '12345fg'
  *****************************************************************************/
}
```

"strcpy — Copy Strings" on page 544
"strncpy — Copy Strings" on page 570
"wcscpy — Copy Wide-Character Strings" on page 715
"wcsncat — Concatenate Wide-Character Strings" on page 723
"wcsncmp — Compare Wide-Character Strings" on page 725
"<wcstr.h>" on page 782

wcspbrk — Locate Wide Characters in String

Format

#include <wcstr.h>
wchar_t *wcspbrk(const wchar_t *_string1_, const wchar_t *_string2_);

Description

Language Level: SAA, XPG4

wcspbrk locates the first occurrence in the string pointed to by _string1_ of any wide character from the string pointed to by _string2_.

Return Value

wcspbrk returns a pointer to the character. If _string1_ and _string2_ have no wide characters in common, wcspbrk returns NULL.

This example uses wcspbrk to find the first occurrence of either a or b in the array _string_.

```
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t  *result;
     wchar_t *string = L"A Blue Danube";
     wchar_t *chars = L"ab";

     result = wcspbrk(string, chars);
     printf("The first occurrence of any of the characters \"%ls\" in "
          "\"%ls\" is \"%ls\"\n", chars, string, result);
    return  0;

   /*************************************************************************
        The output should be similar to:

        The first occurrence of any of the characters "ab" in "A Blue Danube"
       is  "anube"
   *************************************************************************/
}
```

“strchr — Search for Character” on page 537
“strcspn — Compare Strings for Substrings” on page 546
“strpbrk — Find Characters in String” on page 575
“strrchr — Find Last Occurrence of Character in String” on page 581
“strspn — Search Strings” on page 585
“wcschr — Search for Wide Character” on page 709
“wcscmp — Compare Wide-Character Strings” on page 711
“wcscspn — Find Offset of First Wide-Character Match” on page 717
“wcsncmp — Compare Wide-Character Strings” on page 725
“wcsrchr — Locate Wide Character in String” on page 731
“wcsspn — Search Wide-Character Strings” on page 735

wcspbrk

"wcswcs — Locate Wide-Character Substring" on page 747
"<wcstr.h>" on page 782

wcsrchr — Locate Wide Character in String

Format

```
#include <wcstr.h>
wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

Description

Language Level: SAA, XPG4

wcsrchr locates the last occurrence of *character* in the string pointed to by *string*. The terminating wchar_t null character is considered to be part of the string.

Return Value

wcsrchr returns a pointer to the character, or a NULL pointer if *character* does not occur in the string.

This example compares the use of wcschr and wcsrchr. It searches the string for the first and last occurrence of p in the wide character string.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
    wchar_t buf[SIZE] = L"computer program";
   wchar_t  *ptr;
    int ch = 'p';

    /* This illustrates wcschr                                                  */

    ptr = wcschr(buf, ch);
    printf("The first occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr);

    /* This illustrates wscrchr                                                 */

    ptr = wcsrchr(buf, ch);
    printf("The last occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr);
   return  0;

   /****************************************************************************
        The output should be:

        The first occurrence of p in 'computer program' is 'puter program'
        The last occurrence of p in 'computer program' is 'program'
   ****************************************************************************/
}
```

wcsrchr

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcsspn — Search Wide-Character Strings" on page 735
"wcswcs — Locate Wide-Character Substring" on page 747
"wcspbrk — Locate Wide Characters in String" on page 729
"<wcstr.h>" on page 782

wcsrtombs — Convert Wide-Character String to Multibyte String

Format

```
#include <wchar.h>
size_t wcsrtombs (char *dest, const wchar_t **src,
                  size_t len, mbstate_t *ps);
```

Description

Language Level: ANSI 93

wcsrtombs is a restartable version of wcstombs, and performs the same function. It converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters and then stores the converted characters into the array pointed to by *dest*.

Conversion continues up to and including the terminating **wchar_t** null character. The terminating **wchar_t** null character is also stored. Conversion stops earlier if a sequence of bytes does not form a valid multibyte character, or when *len* codes have been stored into the array pointed to by *dest*. Each conversion takes places as if by a call to the wcrtomb function.

wcsrtombs assigns the object pointed to by *src* either a null pointer (if conversion stopped because a terminating null character was reached) or the address just past the last wide character converted. With wcsrtombs, you can begin processing a multibyte string, and then another. Once you have begun processing the second string, you may switch back and continue with the first string. On platforms that support shift states, *ps* represents the initial shift state of the string (0). If you read in only part of the string, wcsrtombs sets *ps* to the string's shift state at the point you stopped. You can then call wcsrtombs again for that string and pass in the updated *ps* value to continue reading where you left off.

Note: Because the Windows code pages do not have shift states, the *ps* parameter is provided only for compatibility with other ANSI/ISO platforms. VisualAge for C++ ignores the value passed for *ps*.

The behavior of wcsrtombs is affected by the LC_CTYPE category of the current locale.

Return Value

wcsrtombs returns the number of bytes in the resulting multibyte character sequence pointed to by *dest*. If *dest* is a null pointer, the value of *len* is ignored and wcsrtombs returns the number of elements required for the converted wide characters.

If the input string contains an invalid wide character, wcsrtombs sets errno to EILSEQ and returns (size_t)−1.

wcsrtombs

This example uses wcsrtombs to convert two wide-character strings to multibyte character strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

#define SIZE    20
#define LOCNAME "ja_jp.ibm-932"

int main(void)
{
    wchar_t             wcs1[] = L"abc";
    wchar_t             wcs2[] = L"A\x8142" L"C";
    const wchar_t *pwcs1 = wcs1;
    const wchar_t *pwcs2 = wcs2;
    mbstate_t         ss1 = 0;
    mbstate_t         ss2 = 0;
   char                 mb1[SIZE],  mb2[SIZE];

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
       exit(1);
    }
    wcsrtombs(mb1, &pwcs1, SIZE, &ss1);
    wcsrtombs(mb2, &pwcs2, SIZE, &ss2);
    printf("The first multibyte string is: <%s>\n", mb1);
    printf("The second multibyte string is: <%s>\n", mb2);
   return  0;

   /*************************************************************************
        The output should be similiar to :

        The first multibyte string is: <abc>
        The second multibyte string is: <A BC>
   *************************************************************************/
}
```

"mblen — Determine Length of Multibyte Character" on page 382
"mbrlen — Calculate Length of Multibyte Character" on page 384
"mbrtowc — Convert Multibyte Character to Wide Character" on page 386
"mbsrtowcs — Convert Multibyte String to Wide-Character String" on page 389
"mbstowcs — Convert Multibyte String to Wide-Character String" on page 391
"wcrtomb — Convert Wide Character to Multibyte Character" on page 706
"wcstombs — Convert Wide-Character String to Multibyte String" on page 743
"<wchar.h>" on page 780

wcsspn — Search Wide-Character Strings

Format

#include <wcstr.h>
size_t wcsspn(const wchar_t *_string1_, const wchar_t *_string2_);

Description

Language Level: SAA, XPG4

wcsspn scans _string1_ for the wide characters contained in _string2_. It stops when it encounters a character in _string1_ that is not in _string2_.

Return Value

wcsspn returns the number of wide characters from _string2_ that it found in _string1_.

This example finds the first occurrence in the array string of a wide character that is not an a, b, or c. Because the string in this example is cabbage, wcsspn returns 5, the index of the segment of cabbage before a character that is not an a, b, or c.

```
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t *string = L"cabbage";
    wchar_t *source = L"abc";
   int  index;

    index = wcsspn(string, L"abc");
    printf("The first %d characters of \"%ls\" are found in \"%ls\"\n", index,
        string,  source);
   return  0;

   /*************************************************************************
        The output should be:

        The first 5 characters of "cabbage" are found in "abc"
   *************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcsrchr — Locate Wide Character in String" on page 731
"wcsspn — Search Wide-Character Strings"
"wcswcs — Locate Wide-Character Substring" on page 747
"wcspbrk — Locate Wide Characters in String" on page 729
"<wcstr.h>" on page 782

wcsstr — Locate Wide-Character Substring

Format

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

Description

Language Level: ANSI 93

wcsstr locates the first occurrence of *wcs2* in *wcs1*. In the matching process, wcsstr ignores the **wchar_t** null character that ends *wcs2*.

The behavior of wcsstr is affected by the LC_CTYPE category of the current locale.

Return Value

wcsstr returns a pointer to the beginning of the first occurrence of *wcs2* in *wcs1*. If *wcs2* does not appear in *wcs1*, wcsstr returns NULL. If *wcs2* points to a wide-character string with zero length, wcsstr returns *wcs1*.

This example uses wcsstr to find the first occurrence of hay in the wide-character string needle in a haystack.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs1 = L"needle in a haystack";
    wchar_t *wcs2 = L"hay";

    printf("result: \"%ls\"\n", wcsstr(wcs1, wcs2));
   return  0;

   /*****************************************************************************
        The output should be similar to :

       result:  "haystack"
   *****************************************************************************/
}
```

"strstr — Locate Substring" on page 587
"wcschr — Search for Wide Character" on page 709
"wcsrchr — Locate Wide Character in String" on page 731
"wcswcs — Locate Wide-Character Substring" on page 747
"<wchar.h>" on page 780

wcstod — Convert Wide-Character String to Double

Format

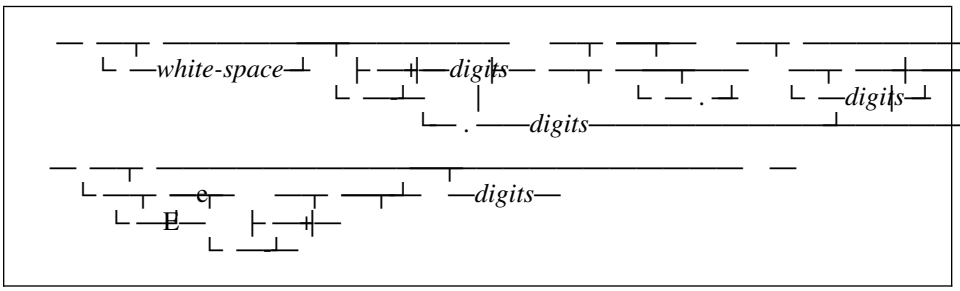
```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Description

Language Level: ANSI 93, XPG4

wcstod converts the wide-character string pointed to by *nptr* to a double value. The *nptr* parameter points to a sequence of characters that can be interpreted as a numerical value of type double. wcstod stops reading the string at the first character that it cannot recognize as part of a number. This character can be the **wchar_t** null character at the end of the string.

wcstod expects *nptr* to point to a string with the following form:



Note: The character used for the decimal point (shown as . in the above diagram) depends on the LC_NUMERIC category of the current locale.

In locales other than the "C" locale, additional implementation-defined subject sequence forms may be accepted.

The behavior of wcstod is affected by the LC_CTYPE category of the current locale.

Return Value

wcstod function returns the converted double value. If no conversion could be performed, wcstod returns 0. If the correct value is outside the range of representable values, wcstod returns +HUGE_VAL or -HUGE_VAL (according to the sign of the value), and sets errno to ERANGE. If the correct value would cause underflow, wcstod returns 0 and sets errno to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

This example uses wcstod to convert the string wcs to a floating-point value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"3.1415926This stopped it";
   wchar_t  *stopwcs;

    printf("wcs = \"%ls\"\n", wcs);
   printf("    wcstod = %f\n", wcstod(wcs, &stopwcs));
   printf("    Stop scanning at \"%ls\"\n", stopwcs);
   return 0;

   /*****************************************************************************
        The output should be similar to :

        wcs = "3.1415926This stopped it"
            wcstod = 3.141593
            Stop scanning at "This stopped it"
   *****************************************************************************/
}
```

"strtod — Convert Character String to Double" on page 590
"strtol — Convert Character String to Long Integer" on page 594
"strtold — Convert String to Long Double" on page 596
"strtoul — Convert String Segment to Unsigned Integer" on page 600
"wcstol — Convert Wide-Character to Long Integer" on page 741
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"<wchar.h>" on page 780

wcstok — Tokenize Wide-Character String

Format

#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);

Description

Language Level: ANSI 93, XPG4

wcstok reads *wcs1* as a series of zero or more tokens and *wcs2* as the set of wide characters serving as delimiters for the tokens in *wcs1*. A sequence of calls to wcstok locates the tokens inside *wcs1*. The tokens can be separated by one or more of the delimiters from *wcs2*. The third argument points to a wide-character pointer that you provide where wcstok stores information necessary for it to continue scanning the same string.

When wcstok is first called for the wide-character string *wcs1*, it searches for the first token in *wcs1*, skipping over leading delimiters. wcstok returns a pointer to the first token.

To read the next token from *wcs1*, call wcstok with NULL as the first parameter (*wcs1*). This NULL parameter causes wcstok to search for the next token in the previous token string. Each delimiter is replaced by a null character to terminate the token.

wcstok always stores enough information in the pointer *ptr* so that subsequent calls, with NULL as the first parameter and the unmodified pointer value as the third, will start searching right after the previously returned token. You can change the set of delimiters (*wcs2*) from call to call.

The behavior of wcstok function is affected by the LC_CTYPE category of the current locale.

Return Value

wcstok function returns a pointer to the first wide character of the token, or a null pointer if there is no token. In later calls with the same token string, strtok returns a pointer to the next token in the string. When there are no more tokens, strtok returns NULL.

wcstok

This example uses wcstok to locate the tokens in the wide-character string str1.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    static wchar_t str1[] = L"?a??b,,,#c";
    static wchar_t str2[] = L"\t \t";
    wchar_t *t, *ptr1, *ptr2;

    t = wcstok(str1, L"?", &ptr1);          /* t points to the token L"a"       */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L",", &ptr1);          /* t points to the token L"?b"      */
    printf("t = '%ls'\n", t);
    t = wcstok(str2, L" \t,", &ptr2);    /* t is a null pointer                 */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"#,", &ptr1);       /* t points to the token L"c"        */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"?", &ptr1);          /* t is a null pointer              */
    printf("t = '%ls'\n", t);
   return  0;

   /*****************************************************************************
        The output should be similar to :

              t = 'a'
              t = '?b'
              t = '(null)'
              t = 'c'
              t = '(null)'
   *****************************************************************************/
}
```

"strtok — Tokenize String" on page 592
"<wchar.h>" on page 780

wcstol — Convert Wide-Character to Long Integer

Format

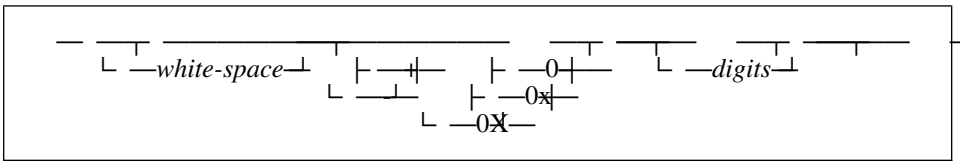
```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description

Language Level: ANSI 93, XPG4

wcstol converts the wide-character string pointed to by *nptr* to a long integer value. *nptr* points to a sequence of wide characters that can be interpreted as a numerical value of type long int. wcstol stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the **wchar_t** null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base*.

When you use wcstol, *nptr* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

In locales other than the "C" locale, additional implementation-defined forms may be accepted.

The behavior of wcstol is affected by the LC_CTYPE category of the current locale.

Return Value

wcstol returns the converted long integer value. If no conversion could be performed, wcstol returns 0. If the correct value is outside the range of representable values, wcstol returns LONG_MAX or LONG_MIN returned (according to the sign of the value), and sets **errno** to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

This example uses wcstol to convert the wide-character string wcs to a long integer value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"10110134932";
    wchar_t  *stopwcs;
    long         l;
    int          base;

    printf("wcs = \"%ls\"\n", wcs);
    for (base=2; base<=8; base*=2) {
        l = wcstol(wcs, &stopwcs, base);
        printf("      wcstol = %ld\n"
                  "      Stopped scan at \"%ls\"\n\n", l, stopwcs);
    }
    return  0;

    /*****************************************************************************
        The output should be similar to :

        wcs = "10110134932"
            wcstol = 45
            Stopped scan at "34932"

            wcstol = 4423
            Stopped scan at "4932"

            wcstol = 2134108
            Stopped scan at "932"

    *****************************************************************************/
}
```

"atol — Convert Character String to Long Integer" on page 60
"strtol — Convert Character String to Long Integer" on page 594
"strtold — Convert String to Long Double" on page 596
"strtoul — Convert String Segment to Unsigned Integer" on page 600
"wcstod — Convert Wide-Character String to Double" on page 737
"wcstoul — Convert Wide-Character String to Unsigned Long" on page 745
"<wchar.h>" on page 780

wcstombs — Convert Wide-Character String to Multibyte String

Format

```
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t n);
```

Description

Language Level: ANSI, SAA, XPG4

wcstombs converts the wide-character string pointed to by *string* into the multibyte array pointed to by *dest*. The conversion stops after *n* bytes in *dest* are filled or after a wide null character is encountered.

Only complete multibyte characters are stored in *dest*. If the lack of space in *dest* would cause a partial multibyte character to be stored, wcstombs stores fewer than *n* bytes and discards the invalid character.

The behavior of wcstombs is affected by the LC_CTYPE category of the current locale.

Return Value

If successful, wcstombs returns the number of bytes converted and stored in *dest*, not counting the terminating null character. The string pointed to by *dest* ends with a null character unless wcstombs returns the value *n*.

If it encounters an invalid wide character, wcstombs returns (size_t)−1.

If the area pointed to by *dest* is too small to contain all the wide characters represented as multibyte characters, wcstombs returns the number of bytes containing complete multibyte characters.

If *dest* is a null pointer, the value of *len* is ignored and wcstombs returns the number of elements required for the converted wide characters.

In this example, wcstombs converts a wide-character string to a multibyte character string twice. The first call converts the entire string, while the second call only converts three characters. The results are printed each time.

wcstombs

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define   SIZE                20

int main(void)
{
    char  dest[SIZE];
     wchar_t *dptr = L"string";
     size_t count = SIZE;
    size_t  length;

     length = wcstombs(dest, dptr, count);
     printf("%d characters were converted.\n", length);
     printf("The converted string is \"%s\"\n\n", dest);

     /* Reset the destination buffer                                          */

     memset(dest, '\0', sizeof(dest));

     /* Now convert only 3 characters                                         */

     length = wcstombs(dest, dptr, 3);
     printf("%d characters were converted.\n", length);
     printf("The converted string is \"%s\"\n", dest);
    return  0;

    /*****************************************************************************
         The output should be:

         6 characters were converted.
         The converted string is "string"

         3 characters were converted.
         The converted string is "str"
    *****************************************************************************/
}
```

"mbstowcs — Convert Multibyte String to Wide-Character String" on page 391
"wcslen — Calculate Length of Wide-Character String" on page 722
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"wctomb — Convert Wide Character to Multibyte Character" on page 753
"<stdlib.h>" on page 775

wcstoul — Convert Wide-Character String to Unsigned Long

Format

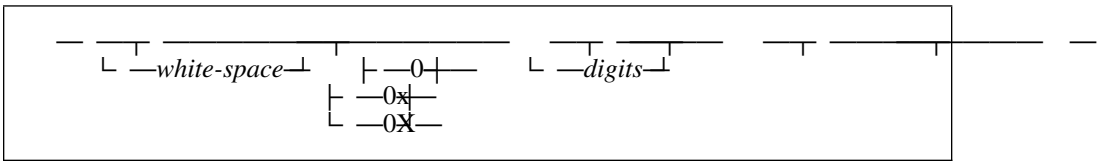
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);

Description

Language Level: ANSI, SAA, XPG4

wcstoul converts the wide-character string pointed to by *nptr* to an unsigned long integer value. *nptr* points to a sequence of wide characters that can be interpreted as a numerical value of type unsigned long int. wcstoul stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the **wchar_t** null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base*.

When you use wcstoul, *nptr* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8 (octal); the prefix 0x or 0X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

In locales other than the "C" locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed and wcstoul stores the value of *nptr* in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The behavior of wcstoul is affected by the LC_CTYPE category of the current locale.

Return Value

wcstoul returns the converted unsigned long integer value. If no conversion could be performed, wcstoul returns 0. If the correct value is outside the range of representable values, wcstoul returns ULONG_MAX and sets **errno** to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

This example uses wcstoul to convert the string wcs to an unsigned long integer value.

```
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
    wchar_t *wcs = L"1000e13 camels";
   wchar_t  *endptr;
    unsigned long int answer;

    answer = wcstoul(wcs, &endptr, BASE);
    printf("The input wide string used:  %ls \n"
               "The unsigned long int produced: %lu\n"
               "The substring of the input wide string that was not"
               " converted to unsigned long:  %ls \n", wcs, answer, endptr);
   return  0;

   /*****************************************************************************
        The output should be similar to :

        The input wide string used:  1000e13 camels
        The unsigned long int produced: 8
        The substring of the input wide string that was not converted to
        unsigned long:  e13 camels
   *****************************************************************************/
}
```

“strtod — Convert Character String to Double” on page 590
“strtol — Convert Character String to Long Integer” on page 594
“strtol — Convert Character String to Long Integer” on page 594
“strtold — Convert String to Long Double” on page 596
“wcstod — Convert Wide-Character String to Double” on page 737
“wcstol — Convert Wide-Character to Long Integer” on page 741
“<wchar.h>” on page 780

wcswcs — Locate Wide-Character Substring

Format

```
#include <wcstr.h>
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

Description

Language Level: SAA, XPG4

wcswcs locates the first occurrence of *string2* in the wide-character string pointed to by *string1*. In the matching process, wcswcs ignores the wchar_t null character that ends *string2*.

Return Value

wcswcs returns a pointer to the located string or NULL if the string is not found. If *string2* points to a string with zero length, wcswcs returns *string1*.

This example finds the first occurrence of the wide character string pr in buffer1.

```
#include <stdio.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
   wchar_t  *ptr;
    wchar_t *wch = L"pr";

    ptr = wcswcs(buffer1, wch);
    printf("The first occurrence of %ls in '%ls' is '%ls'\n", wch, buffer1, ptr);
   return  0;

  /*****************************************************************************
       The output should be:

       The first occurrence of pr in 'computer program' is 'program'
  *****************************************************************************/
}
```

"strchr — Search for Character" on page 537
"strcspn — Compare Strings for Substrings" on page 546
"strpbrk — Find Characters in String" on page 575
"strrchr — Find Last Occurrence of Character in String" on page 581
"strspn — Search Strings" on page 585
"wcschr — Search for Wide Character" on page 709
"wcscspn — Find Offset of First Wide-Character Match" on page 717
"wcspbrk — Locate Wide Characters in String" on page 729
"wcsrchr — Locate Wide Character in String" on page 731
"wcsspn — Search Wide-Character Strings" on page 735
"<wcstr.h>" on page 782

wcswidth — Determine Display Width of Wide-Character String

Format

#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);

Description

Language Level: XPG4

wcswidth determines the number of printing positions occupied on a display device by a graphic representation of *n* wide characters in the string pointed to by *wcs* (or fewer than *n* wide characters, if a null wide character is encountered before *n* characters have been exhausted). The number is independent of its location on the device.

The behavior of wcswidth is affected by the LC_CTYPE category.

Return Value

wcswidth returns the number of printing positions occupied by the wide-character string. If *wcs* points to a null wide character, wcswidth returns 0. If any wide character in *wcs* is not a printing character, wcswidth returns −1.

Note: Under VisualAge for C++, the printing width is 0 for a null character, 1 for each single-byte character, and 2 for each double-byte character.

This example uses wcswidth to calculate the display width of ABC.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"ABC";

    printf("wcs has a width of: %d\n", wcswidth(wcs,3));
   return  0;

   /*****************************************************************************
        The output should be similar to :

        wcs has a width of: 3
   *****************************************************************************/
}
```

"wcwidth — Determine Display Width of Wide Character" on page 758
"<wchar.h>" on page 780

wcsxfrm — Transform Wide-Character String

Format

```
#include <wchar.h>
size_t wcsxfrm(wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

Description

Language Level: ANSI 93, XPG4

wcsxfrm transforms the wide-character string pointed to by *wcs2* to values that represent character collating weights and places the resulting wide-character string into the array pointed to by *wcs1*. The transformation is determined by the program's locale. The transformed string may not be displayable or printable but, can be used with the wcscmp function.

The transformation is such that if wcscmp were applied to two transformed wide-character strings, the results would be the same as applying the wcscoll function to the two corresponding untransformed strings.

No more than *n* elements are placed into the resulting array pointed to by *wcs1*, including the terminating null wide character. If *n* is 0, *wcs1* can be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The behavior of wcsxfrm is controlled by the LC_COLLATE category of the current locale.

Return Value

wcsxfrm returns the length of the transformed wide-character string (excluding the terminating null wide character). If the value returned is *n* or more, the contents of the array pointed to by *wcs1* are indeterminate.

If *wcs1* is a null pointer, wcsxfrm returns the number of elements required to contain the transformed wide string.

If *wcs2* contains invalid wide characters, wcsxfrm returns (size_t)–1. The transformed values of the invalid characters are either less than or greater than the transformed values of valid wide characters, depending on the option chosen for the particular locale definition.

If *wcs2* contains wide characters outside the domain of the collating sequence. wcsxfrm sets **errno** to EILSEQ.

This example uses wcsxfrm to transform two different strings with the same collating weight. It then uses wcscmp to compare the new strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

#if (1 == __TOS_OS2__)
    #define LOCNAME "da_dk.ibm-865"        /* OS/2 name              */
    wchar_t *string1 = L"str\xA0ng1a";
    wchar_t *string2 = L"strang1\x83";
#else
    #define LOCNAME "da_dk.ibm-1252"       /* Windows name           */
    wchar_t *string1 = L"str\xE0ng1a";
    wchar_t *string2 = L"strang1\xE2";
#endif

int main(void)
{
    wchar_t *newstring1, *newstring2;
    size_t    length1, length2, pw1, pw2;

    if (NULL == setlocale(LC_ALL, LOCNAME)) {
        printf("Locale \"%s\" could not be loaded\n", LOCNAME);
        exit(1);
    }
    length1 = wcsxfrm(NULL, string1, 0);
    pw1 = (wcslen(string1)/2);
    length2 = wcsxfrm(NULL, string2, 0);
    pw2 = (wcslen(string2)/2);
    if (NULL == (newstring1 = (wchar_t*)calloc(length1 + 1, sizeof(wchar_t))) ||
          NULL == (newstring2 = (wchar_t*)calloc(length2 + 1, sizeof(wchar_t)))) {
        printf("insufficient  memory\n");
        exit(1);
    }
    /* Get primary weight of each string                                    */
    if ((wcsxfrm(newstring1, string1, pw1 + 1) != length1) ||
        (wcsxfrm(newstring2, string2, pw2 + 1) != length2)) {
        printf("error in string processing\n");
        exit(1);
    }
    if (0 != wcscmp(newstring1, newstring2))
        printf("wrong  results\n");
    else
        printf("correct  results\n");
    return  0;

    /*************************************************************************
        The output should be similar to :

        correct  results
    *************************************************************************/
}
```

"strxfrm — Transform String" on page 605
"wcscmp — Compare Wide-Character Strings" on page 711
"wcscoll — Compare Wide-Character Strings" on page 713

"<wchar.h>" on page 780

wctob — Convert Wide Character to Byte

Format

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);
```

Description

Language Level: ANSI 93

wctob determines whether *wc* corresponds to a member of the extended character set, whose multibyte character has a length of 1 byte

The behavior of wctob is affected by the LC_CTYPE category of the current locale.

Return Value

If *c* corresponds to a multibyte character with a length of 1 byte, wctob returns the single-byte representation. Otherwise, wctob returns EOF.

This example uses wctob to test if the wide character A is a valid single-byte character.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    if (wctob(wc) == wc)
        printf("%lc is a valid single byte character\n", wc);
   else
        printf("%lc is not a valid single byte character\n", wc);
   return  0;

   /*************************************************************************
        The output should be similar to :

        A is a valid single byte character
   *************************************************************************/
}
```

"mbtowc — Convert Multibyte Character to Wide Character" on page 393
"wctomb — Convert Wide Character to Multibyte Character" on page 753
"wcstombs — Convert Wide-Character String to Multibyte String" on page 743
"<wchar.h>" on page 780

wctomb — Convert Wide Character to Multibyte Character

Format

```
#include <stdlib.h>
int wctomb(char *string, wchar_t wc);
```

Description

Language Level: ANSI, SAA, XPG4

wctomb converts the wide character *wc* into a multibyte character and stores it in the location pointed to by *string*. wctomb stores a maximum of MB_CUR_MAX characters in *string*.

The behavior of wctomb is affected by the LC_CTYPE category of the current locale.

Return Value

wctomb returns the length in bytes of the multibyte character. If *wc* is not a valid multibyte character, wctomb returns −1.

If *string* is a NULL pointer, wctomb returns 0.

Note: On platforms that support shift states, wctomb can also return a nonzero value to indicate that the multibyte encoding is state dependent. Because VisualAge for C++ does not support state-dependent encoding, wctomb always returns 0 when *string* is NULL.

This example calls wctomb to convert the wide character c to a multibyte character.

```
#include <stdio.h>
#include <stdlib.h>
#include <wcstr.h>

#define   SIZE              40

int main(void)
{
     static char buffer[SIZE];
     wchar_t wch = L'c';
    int  length;

     length = wctomb(buffer, wch);
     printf("The number of bytes that comprise the multibyte ""character is %i\n",
         length);
     printf("And the converted string is \"%s\"\n", buffer);
    return  0;

   /**************************************************************************
         The output should be:

         The number of bytes that comprise the multibyte character is 1
         And the converted string is "c"
   **************************************************************************/
}
```

"mbtowc — Convert Multibyte Character to Wide Character" on page 393

wctomb

"wcrtomb — Convert Wide Character to Multibyte Character" on page 706
"wcslen — Calculate Length of Wide-Character String" on page 722
"wcsrtombs — Convert Wide-Character String to Multibyte String" on page 733
"wcstombs — Convert Wide-Character String to Multibyte String" on page 743
"wctob — Convert Wide Character to Byte" on page 752
"<stdlib.h>" on page 775

wctype — Get Handle for Character Property Classification

Format

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

Language Level: ANSI 93, XPG4

wctype returns a handle for the specified character *class* from the LC_CTYPE category. You can then use this handle with the iswctype function to determine if a given wide character belongs to that *class*.

The following strings correspond to the standard (basic) character classes or properties:

"alnum"	"cntrl"	"lower"	"space"
"alpha"	"digit"	"print"	"upper"
"blank"	"graph"	"punct"	"xdigit"

These classes are described in “isalnum to isxdigit — Test Integer Value” on page 315 and “iswalnum to iswxdigit — Test Wide Integer Value” on page 327.

You can also give the name of a user-defined class, as specified in a locale definition file, as the *property* argument.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale.

Return Value

wctype returns a value of type wctype_t that represents the property and can be used in calls to iswctype. If the given *property* is not valid for the current locale (LC_CTYPE category), wctype returns 0.

Values returned by wctype are valid until a call to setlocale that modifies the LC_CTYPE category.

This example uses wctype to return each standard property, and calls iswctype to test a wide character against each property.

```
#include <wctype.h>
#include <stdio.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
    int  wc;

     for (wc = 0; wc <= UPPER_LIMIT; wc++) {
          printf("%#4x ", wc);
          printf("%c", iswctype(wc, wctype("print"))    ? wc        : ' ');
          printf("%s", iswctype(wc, wctype("alnum"))    ? " AN" : "      ");
          printf("%s", iswctype(wc, wctype("alpha"))    ? " A " : "      ");
          printf("%s", iswctype(wc, wctype("blank"))    ? " B " : "      ");
          printf("%s", iswctype(wc, wctype("cntrl"))    ? " C " : "      ");
          printf("%s", iswctype(wc, wctype("digit"))    ? " D " : "      ");
          printf("%s", iswctype(wc, wctype("graph"))    ? " G " : "      ");
          printf("%s", iswctype(wc, wctype("lower"))    ? " L " : "      ");
          printf("%s", iswctype(wc, wctype("punct"))    ? " PU" : "      ");
          printf("%s", iswctype(wc, wctype("space"))    ? " S " : "      ");
          printf("%s", iswctype(wc, wctype("print"))    ? " PR" : "      ");
          printf("%s", iswctype(wc, wctype("upper"))    ? " U " : "      ");
          printf("%s", iswctype(wc, wctype("xdigit")) ? " X " : "      ");
         putchar('\n');
    }
    return  0;
```

```
/*****************************************************************************
     The output should be similar to :
      :
     0x1f                C
     0x20            B                    S   PR
     0x21  !                  G       PU      PR
     0x22  "                  G       PU      PR
     0x23  #                  G       PU      PR
     0x24  $                  G       PU      PR
     0x25  %                  G       PU      PR
     0x26  &                  G       PU      PR
     0x27  '                  G       PU      PR
     0x28  (                  G       PU      PR
     0x29  )                  G       PU      PR
     0x2a  *                  G       PU      PR
     0x2b  +                  G       PU      PR
     0x2c  ,                  G       PU      PR
     0x2d  -                  G       PU      PR
     0x2e  .                  G       PU      PR
     0x2f  /                  G       PU      PR
      0x30 0 AN           D   G              PR      X
      0x31 1 AN           D   G              PR      X
      0x32 2 AN           D   G              PR      X
      0x33 3 AN           D   G              PR      X
      0x34 4 AN           D   G              PR      X
      0x35 5 AN           D   G              PR      X
      :
*****************************************************************************/
}
```

"iswalnum to iswxdigit — Test Wide Integer Value" on page 327
"iswctype — Test for Character Property" on page 333
"<wchar.h>" on page 780

wcwidth — Determine Display Width of Wide Character

Format

```
#include <wchar.h>
int wcwidth (const wint_t wc);
```

Description

Language Level: XPG4

wcwidth determines the number of printing positions that a graphic representation of *wc* occupies on a display device. Each of the printing wide characters occupies its own number of printing positions on a display device. The number is independent of its location on the device.

The behavior of wcwidth is affected by the LC_CTYPE category.

Return Value

wcwidth returns the number of printing positions occupied by *wc*. If *wc* is a null wide character, wcwidth returns 0. If *wc* is not a printing wide character, *wc* returns −1.

Note: Under VisualAge for C++, the printing width is 0 for a null character, 1 for a single-byte character, and 2 for a double-byte character.

This example determines the printing width for the wide character A.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    printf("%lc has a width of %d\n", wc, wcwidth(wc));
   return  0;

   /*************************************************************************
       The output should be similar to :

       A has a width of 1
   *************************************************************************/
}
```

"wcswidth — Determine Display Width of Wide-Character String" on page 748
"<wchar.h>" on page 780

write — Writes from Buffer to File

Format

#include <io.h>
int write(int *handle*, const void **buffer*, unsigned int *count*);

Description

Language Level: XPG4, Extension

write writes *count* bytes from *buffer* into the file associated with *handle*. The write operation begins at the current position of the file pointer associated with the given file. If the file is open for appending, the operation begins at the end of the file. After the write operation, the file pointer is increased by the number of bytes actually written.

If the given file was opened in text mode, each line feed character is replaced with a carriage return/line feed pair in the output. The replacement does not affect the return value.

Note: In earlier releases of VisualAge C++, write began with an underscore (_write). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, VisualAge for C++ will map _write to write for you.

Return Value

write returns the number of bytes moved from the buffer to the file. The return value may be positive but less than *count*. A return value of -1 indicates an error, and errno is set to one of the following values:

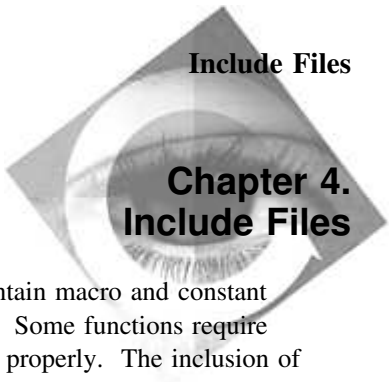
Value	Meaning
EBADF	The file handle is not valid, or the file is not open for writing.
ENOSPC	No space is left on the device.
EOS2ERR	The call to the operating system was not successful.

This example writes the contents of the character array buffer to the output file whose handle is fh.

write

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

#define FILENAME    "write.dat"

int main(void)
{
    int  fh;
    char  buffer[20];

     memset(buffer, 'a', 20);                    /* initialize storage                    */
     printf("\nCreating " FILENAME ".\n");
     system("echo Sample Program > " FILENAME);
     if (-1 == (fh = open(FILENAME, O_RDWR|O_APPEND))) {
          perror("Unable to open " FILENAME);
          return  EXIT_FAILURE;
     }
     if (5 != write(fh, buffer, 5)) {
          perror("Unable to write to " FILENAME);
          close(fh);
          return  EXIT_FAILURE;
     }
     printf("Successfully appended 5 characters.\n");
    close(fh);
    return  0;

    /*************************************************************************
          The program should create a file "write.dat" containing:

          Sample  Program
          aaaaa

          The output should be:

          Creating  write.dat.
          Successfully appended 5 characters.
    *************************************************************************/
}
```

"creat — Create New File" on page 100
"fread — Read Items" on page 236
"fwrite — Write Items" on page 263
"lseek — Move File Pointer" on page 365
"open — Open File" on page 418
"read — Read Into Buffer" on page 450
"_sopen — Open Shared File" on page 516
"_tell — Get Pointer Position" on page 619
"<io.h>" on page 764



Chapter 4. Include Files

The include files provided with the runtime library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from include files to work properly. The inclusion of files is optional, as long as the necessary statements from the files are coded directly into the source.

Note: If you change the default calling convention (using one of the /M options), you must include the appropriate header files with the declarations of the library functions that you use.

This section describes each include file, explains its contents, and lists the functions that are declared in the file.

<assert.h>

The <assert.h> include file defines the assert macro. You must include assert.h when you use assert.

The definition of assert is in an **#ifndef** preprocessor block. If you have not defined the identifier **NDEBUG** through a **#define** directive or on the compiler command line, the assert macro tests a given expression (the assertion). If the assertion is false, the system prints a message to **stderr** and an abort signal is raised for the program.

If **NDEBUG** is defined, assert is defined to do nothing. You can suppress program assertions by defining **NDEBUG**.

<builtin.h>

The <builtin.h> include file declares the following built-in and intrinsic functions:

_alloca	_facos	_fsincos	_lrotl	_rotl
_clear87	_fasin	_fyl2x	_llrotl	_rotr
_control87	_fcos	_fyl2xp1	_lrotr	_srotl
_crotl	_fcossin	_f2xm1	_llrotr	_srotr
_crotr	_fpatan	_inp	__lxchg	__sxchg
__cxchg	_fptan	_inpd	_outp	_status87
_disable	_fsin	_inpw	_outpd	
_enable	_fsqrt	_interrupt	_outpw	

© Copyright IBM Corp. 1992, 1996

_clear87, _control87, and _status87 are also defined in <float.h>. _alloca is also defined in "<stdlib.h>" on page 775 and "<malloc.h>" on page 769. The functions _inp, _inpd, _inpw, _outp, _outpd, and _outpw are also defined in <conio.h>.

<builtin.h> also includes a declaration for the type **size_t**.

<conio.h>

The <conio.h> include file defines the functions that involve screen and console input and output:

_cgets	_cscanf	_inp	_kbhit	_outpw
_cprintf	_getch	_inpd	_outp	_putch
_cputs	_getche	_inpw	_outpd	_ungetch

<ctype.h>

The <ctype.h> include file defines functions used in character classification. The functions defined in <ctype.h> are:

isalnum	iscntrl	isprint	islower	tolower
isalpha	isdigit	ispunct	isupper	toupper
isblank	isgraph	isspace	isxdigit	

In extended mode, <ctype.h> also includes definitions for the following VisualAge for C++ extensions:

isascii	_iscsymf	_toascii	_tolower	_toupper
_iscsym				

In the VisualAge for C++ compiler, the <ctype.h> functions are all implemented as macros.

<direct.h>

The <direct.h> include file defines functions that control directories and drives. The functions defined in <direct.h> are:

chdir	_getcwd	_getdrive	mkdir	rmdir
_chdrive	_getdcwd			

<errno.h>

The <errno.h> include file defines symbolic macro names, such as EDOM and ERANGE, for runtime errors, and a modifiable lvalue having type **int** called **errno**. It also defines the global variable **_doserrno**, which is determined by the Windows error code when an operating system error occurs.

See the online *User's Guide* for a list of the runtime error messages and the **errno** values associated with each message.

Note: If you are going to test the value of **errno** after library function calls, first set it to 0, because its value may not be reset during the call.

The definitions of **errno** and **_doserrno** are also provided in <stdlib.h> on page 775.

<fcntl.h>

The <fcntl.h> include file defines constants used by the open and _sopen functions. Definitions for these two functions are included in "<io.h>" on page 764. Additional definitions for _sopen are provided in "<share.h>" on page 773.

<float.h>

The <float.h> include file defines constants that specify the ranges of floating-point data types, for example, the maximum number of digits for objects of type **double** or the minimum exponent for objects of type **float**.

In extended mode, <float.h> also defines the macros that represent infinity and NaN (not-a-number) values, and defines the following functions that manipulate the floating-point control and status words, and for the constants that they use:

_clear87 _control87 _fpreset _status87

<iconv.h>

The <iconv.h> include file declares the iconv_open, iconv, and iconv_close functions to convert characters from one character set definition to another. The ICONV utility also uses these functions.

iconv.h also declares the **iconv_t** type, which is a pointer to an object capable of storing the information about the opened converters used.

Include Files

<io.h>

The <io.h> include file defines the following file handle and low-level input and output functions:

access	dup	isatty	remove	_tell
chmod	dup2	lseek	rename	umask
_chsize	__eof	open	_setmode	unlink
close	_filelength	read	_sopen	write
creat				

Two additional functions, fstat and stat, are defined in <sys\stat.h>.

Constants required by the open and _sopen functions are provided in <fcntl.h>. Additional constants used by _sopen are defined in <share.h>.

The unlink function is also defined in <stdio.h>.

<langinfo.h>

The <langinfo.h> include file declares the nl_langinfo function. The include file also defines the macros that, in turn, define constants that are used to identify the information queried in the current locale, and the **nl_item** type, which is the type of the constants. The following macros are defined:

ABDAY_1	Abbreviated first day of the week
ABDAY_2	Abbreviated second day of the week
ABDAY_3	Abbreviated third day of the week
ABDAY_4	Abbreviated fourth day of the week
ABDAY_5	Abbreviated fifth day of the week
ABDAY_6	Abbreviated sixth day of the week
ABDAY_7	Abbreviated seventh day of the week
ABMON_1	Abbreviated first month
ABMON_2	Abbreviated second month
ABMON_3	Abbreviated third month
ABMON_4	Abbreviated fourth month
ABMON_5	Abbreviated fifth month
ABMON_6	Abbreviated sixth month
ABMON_7	Abbreviated seventh month
ABMON_8	Abbreviated eighth month
ABMON_9	Abbreviated ninth month
ABMON_10	Abbreviated tenth month
ABMON_11	Abbreviated eleventh month
ABMON_12	Abbreviated twelfth month
AM_STR	String for morning

CODESET	Current encoded character set of the process
CRNCYSTR	Currency symbol
D_FMT	String for formatting date
D_T_FMT	String for formatting date and time
DAY_1	Name of the first day of the week
DAY_2	Name of the second day of the week
DAY_3	Name of the third day of the week
DAY_4	Name of the fourth day of the week
DAY_5	Name of the fifth day of the week
DAY_6	Name of the sixth day of the week
DAY_7	Name of the seventh day of the week
MON_1	Name of the first month
MON_2	Name of the second month
MON_3	Name of the third month
MON_4	Name of the fourth month
MON_5	Name of the fifth month
MON_6	Name of the sixth month
MON_7	Name of the seventh month
MON_8	Name of the eighth month
MON_9	Name of the ninth month
MON_10	Name of the tenth month
MON_11	Name of the eleventh month
MON_12	Name of the twelfth month
NOEXPR	Negative response expression
PM_STR	String for afternoon
RADIXCHAR	Radix character
T_FMT	String for formatting time
T_FMT_AMPM	String for morning or afternoon time format
THOUSEP	Separator for thousands
YESEXPR	Affirmative response expression

For more information about the effect of locale, see “setlocale — Set Locale” on page 499, “<locale.h>” on page 766, individual functions, and the introduction to locale in the *Programming Guide*.

<lc_core.h>

The <lc_core.h> include file declares the LOCALDEF utility. Do not include it in your source code.

The LOCALDEF utility is described in the *User's Guide*. For more information about locales in general, see the introduction to locale in the *Programming Guide*.

Include Files

<limits.h>

The <limits.h> include file defines constants that specify the ranges of integer and character data types, such as the maximum value for an object of type **char**.

<localdef.h>

The <localdef.h> include file declares the LOCALDEF utility. Do not include it in your source code.

The LOCALDEF utility is described in the *User's Guide*. For more information about locales in general, see the introduction to locale in the *Programming Guide*.

<locale.h>

The <locale.h> include file declares the localdtconv, localeconv, and setlocale library functions, which are useful for changing the C locale when you are creating applications for international users. <locale.h> also declares the **lconv** structure.

The table below shows the elements of the **lconv** structure as well as the defaults for the C locale.

Table 8 (Page 1 of 3). Elements of the lconv Structure

Element	Purpose of Element	Default
char *decimal_point	Decimal-point character used to format nonmonetary quantities.	"."
char *thousands_sep	Character used to separate groups of digits to the left of the decimal-point character in formatted nonmonetary quantities.	""
char *grouping	String indicating the size of each group of digits in formatted nonmonetary quantities. The value of each character in the string determines the number of digits in a group. A value of CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""

Table 8 (Page 2 of 3). Elements of the lconv Structure

Element	Purpose of Element	Default
char *int_curr_symbol	International currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) is the character used to separate the international currency symbol from the monetary quantity.	""
char *currency_symbol	Local currency symbol of the current locale.	""
char *mon_decimal_point	Decimal-point character used to format monetary quantities.	"."
char *mon_thousands_sep	Separator for digits in formatted monetary quantities.	""
char *mon_grouping	String indicating the size of each group of digits in formatted monetary quantities. The value of each character in the string determines the number of digits in a group. A value of CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *positive_sign	String indicating the positive sign used in monetary quantities.	""
char *negative_sign	String indicating the negative sign used in monetary quantities.	""
char int_frac_digits	The number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.	UCHAR_MAX
char frac_digits	Number of digits to the right of the decimal place in monetary quantities.	UCHAR_MAX
char p_cs_precedes	1 if the currency_symbol precedes the value for a nonnegative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char p_sep_by_space	1 if the currency_symbol is separated by a space from the value of a nonnegative formatted monetary quantity; 0 if it does not; 2 if a space separates the symbol and the sign string—if adjacent.	UCHAR_MAX

Table 8 (Page 3 of 3). Elements of the lconv Structure

Element	Purpose of Element	Default
char n_cs_precedes	1 if the currency_symbol precedes the value for a negative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char n_sep_by_space	1 if the currency_symbol is separated by a space from the value of a negative formatted monetary quantity; 0 if it does not; 2 if a space separates the symbol and the sign string—if adjacent.	UCHAR_MAX
char p_sign_posn	Value indicating the position of the positive_sign for a nonnegative formatted monetary quantity.	UCHAR_MAX
char n_sign_posn	Value indicating the position of the negative_sign for a negative formatted monetary quantity.	UCHAR_MAX
char *left_parenthesis	Negative-valued monetary symbol.	""
char *right_parenthesis	Negative-valued monetary symbol.	""
char *debit_sign	Debit_sign characters.	""
char *credit_sign	Credit_sign characters.	""

<locale.h> declares the **dtconv** structure:

```
struct dtconv {
        char *abbrev_month_names[12];   /* Abbreviated month names      */
        char *month_names[12];           /* full month names             */
        char *abbrev_day_names[7];      /* Abbreviated day names        */
        char *day_names[7];              /* full day names               */
        char *date_time_format;         /* date and time format         */
        char *date_format;               /* date format                  */
        char *time_format;               /* time format                  */
        char *am_string;                 /* AM string                    */
        char *pm_string;                 /* PM string                    */
        char *time_format_ampm;          /* long date format             */
};
```

The information in each field is equivalent to calling nl_langinfo with these keywords:

Keyword	Element
ABMON_*nn*	abbrev_month_names
MON_*nn*	month_names
ABDAY_*nn*	abbrev_day_names
DAY_*nn*	day_names
D_T_FMT	date_time_format
D_FMT	date_format
T_FMT	time_format
AM_STR	am_string
PM_STR	pm_string
T_FMT_AMPM	time_format_ampm

Where *nn* is the number corresponding to the day or the month. For example, DAY_7 is the seventh day and ABMON_12 is the abbreviated twelfth month.

<locale.h> also contains definitions for the following macros:

LC_ALL	LC_NUMERIC
LC_COLLATE	LC_TIME
LC_CTYPE	LC_SYNTAX
LC_MESSAGES	LC_MONETARY
LC_TOD	NULL

The aspects of a program related to national language, or to cultural characteristics (such as time zone, currency symbols, and sorting order of characters), can be customized at run time using different locales, to suit users' requirements at those locales. The methods for doing so are discussed in the internationalization chapters of the *Programming Guide*.

<malloc.h>

The <malloc.h> include file defines the following memory allocation functions:

_alloca	_debug_heapmin	free	_heap_walk
calloc	_debug_realloc	_heapchk	malloc
_debug_calloc	_dump_allocated_delta	_heapmin	_msize
_debug_free	_heap_check	_heapset	realloc
_debug_malloc			

It also includes a definition for the type **size_t**.

calloc, free, malloc, and realloc are also defined in <stdlib.h>, as are _alloca and _heapmin.

Include Files

Heap-specific versions of the memory management functions are defined in <umalloc.h>.

Note: To use the debug functions, you must compile with the debug memory (/Tm) option. See "Differentiating between Memory Management Functions" on page 23 for more information about the different types of memory management functions.

<malloc.h> also defines a number of **far** and **near** pointer macros to the corresponding standard library function. These macros are:

_fcalloc	_ffree
_fheapmin	_fmalloc
_frealloc	_ncalloc
_nfree	_nheapmin
_nmalloc	_nrealloc

These macros are also defined in <stdlib.h>.

<math.h>

The <math.h> include file declares all the floating-point math functions:

acos	cos	floor	log	sqrt
asin	cosh	fmod	log10	tan
atan	erf	frexp	modf	tanh
atan2	erfc	gamma	pow	
Bessel	exp	hypot	sin	
ceil	fabs	ldexp	sinh	

Note: The Bessel functions are a group of functions named _j0, _j1, _jn, _y0, _y1, and _yn.

<math.h> also includes definitions for the following VisualAge for C++ extensions:

- _atold
- _cabs
- _matherr

The _atold function is also defined in "<stdlib.h>" on page 775.

<math.h> defines the macros HUGE, _LHUGE, HUGE_VAL, and _LHUGE_VAL, which expand to a positive **double** expression or to infinity.

You can define a macro _FP_INLINE to inline the functions sin, cos, tan, atan, acos and asin. If you use _FP_INLINE to inline the functions, you cannot use the functions defined in <complex.h>, or an error message is generated.

For all mathematical functions, a *domain error* occurs when an input argument is outside the range of values allowed for that function. In the event of a domain error, **errno** is set to the value of **EDOM**.

A range error occurs if the result of the function cannot be represented in a **double** value. If the magnitude of the result is too large (overflow), the function returns the positive or negative value of the macro **HUGE_VAL**, and sets **errno** to **ERANGE**. If the result is too small (underflow), the function returns zero.

<memory.h>

The <memory.h> include file defines the following buffer manipulation functions:

memccpy	memchr
memcmp	memcpy
memicmp	memmove
memset	

Definitions of these functions are also provided in"<string.h>" on page 777 .

The <memory.h> include file also defines a number of **far** and **near** pointer macros to the corresponding standard library function. These macros are:

_fmemccpy	_fmemchr
_fmemcmp	_fmemcpy
_fmemicmp	_fmemset

These macros are also defined in <string.h>.

<monetary.h>

The <monetary.h> include file declares the strfmon function.

For more information about the effect of locale, see "setlocale — Set Locale" on page 499, "<locale.h>" on page 766, individual functions, and the introduction to locale in the *Programming Guide*.

<nl_types.h>

The <nl_types.h> include file defines the **nl_item** type for the nl_langinfo function.

For more information about the effect of locale, see "setlocale — Set Locale" on page 499, "<locale.h>" on page 766, individual functions, and the introduction to locale in the *Programming Guide*.

<process.h>

The <process.h> include file defines the process control functions:

abort	execle	execvp	_spawnl	_spawnve
_beginthread	execlp	_execvpe	_spawnle	_spawnvp
_cwait	_execlpe	exit	_spawnlp	_spawnvpe
_endthread	execv	_exit	_spawnlpe	system
execl	execve	getpid	_spawnv	

The definitions of _beginthread, _endthread, abort, exit, and system are also provided in <stdlib.h>.

<regex.h>

The <regex.h> include file declares the type **size_t** and defines the following regular expression functions:

regcomp	regerror
regexec	regfree

<regex.h> also declares the **regex_t** type, which is capable of storing a compiled regular expression, and the following macros:

REG_EXTENDED
: REG_ICASE
REG_NEWLINE
REG_NOSUB Values of the *cflags* parameter of the regcomp function

REG_NOTBOL
: REG_NOTEOL Values of the *eflags* parameter of the regexec function

REG_* : Values of the *errcode* parameter of the regerror function

<search.h>

The <search.h> include file defines the following search functions:

bsearch	lfind
lsearch	qsort

It also defines the type **size_t**.

Definitions for bsearch and qsort are also provided in <stdlib.h>.

<setjmp.h>

The <setjmp.h> include file declares the setjmp macro and longjmp function. It also defines a buffer type, jmp_buf, that the setjmp macro and longjmp function use to save and restore the program state.

<share.h>

The <share.h> include file defines constants used by the following functions:

creat	fdopen
open	_sopen

The definitions for the above functions are also included in <io.h>. Additional constants for open and _sopen are provided in <fcntl.h>.

<signal.h>

The <signal.h> include file defines the values for signals and declares the signal and raise functions.

<signal.h> also defines the following macros:

SIGABRT	SIG_ERR	SIGILL	SIGTERM	SIGUSR3
SIGBREAK	SIGFPE	SIGINT	SIGUSR1	
SIG_DFL	SIG_IGN	SIGSEGV	SIGUSR2	

<signal.h> also defines the type **sig_atomic_t**.

<stdarg.h>

The <stdarg.h> include file defines macros that allow you access to arguments in functions with variable-length argument lists: va_arg, va_start, and va_end. <stdarg.h> also defines the type **va_list**.

<stddef.h>

The <stddef.h> include file defines the commonly used pointers, variables, and types, from **typedef** statements, as listed below:

ptrdiff_t **typedef** for the type of the difference of two pointers

size_t **typedef** for the type of the value returned by **sizeof**

wchar_t **typedef** for a wide character constant.

Include Files

<stddef.h> also defines the macros NULL and offsetof. NULL is a pointer that is guaranteed not to point to a data object. NULL is also defined in <locale.h>. The offsetof macro expands to the number of bytes between a structure member and the start of the structure. The offsetof macro has the form:

offsetof(*structure_type*, *member*)

In extended mode, <stddef.h> also contains definitions of the global variables **errno** and **_threadid**. **errno** is also defined in <stdlib.h> for extended mode, and in <errno.h> in ANSI and SAA modes.

<stdio.h>

The <stdio.h> include file defines constants, macros, and types, and declares stream input and output functions. The stream I/O functions are:

clearerr	fprintf	fwrite	remove	tmpnam
fclose	fputc	getc	rename	ungetc
feof	fputs	getchar	rewind	vfprintf
ferror	fread	gets	scanf	vprintf
fflush	freopen	perror	setbuf	vsprintf
fgetc	fscanf	printf	setvbuf	
fgetpos	fseek	putc	sprintf	
fgets	fsetpos	putchar	sscanf	
fopen	ftell	puts	tmpfile	

In extended mode, <stdio.h> also defines the following extensions:

_fcloseall	_fputchar	tempnam
fdopen	_flushall	unlink
_fgetchar	_rmtmp	
fileno	_set_crt_msg_handle	

<stdio.h> also defines the macros listed below. You can use these constants in your programs, but you should not alter their values.

BUFSIZ Specifies the buffer size to be used by the setbuf library function when you are allocating buffers for stream I/O. This value establishes the size of system-allocated buffers and is used with setbuf.

EOF The value returned by an I/O function when the end of the file (or in some cases, an error) is found.

FOPEN_MAX The number of files that can be opened simultaneously.

FILENAME_MAX The longest file name supported. If there is no reasonable limit, FILENAME_MAX will be the recommended size.

L_tmpnam	The size of the longest temporary name that can be generated by the tmpnam function.
P_tmpdir	Indicates the default directory to be used by the tempnam function.
TMP_MAX	The minimum number of unique file names that can be generated by the tmpnam function.
NULL	A pointer guaranteed not to point to a data object. NULL is also defined in <locale.h>.

The **FILE** structure type is defined in <stdio.h>. Stream I/O functions use a pointer to the **FILE** type to get access to a given stream. The system uses the information in the **FILE** structure to maintain the stream.

The C standard streams **stdin**, **stdout**, and **stderr** are also defined in <stdio.h>.

The macros SEEK_CUR, SEEK_END, and SEEK_SET expand to integral constant expressions and can be used as the third argument to fseek.

The macros _IOFBF, _IOLBF, and _IONBF expand to integral constant expressions with distinct values suitable for use as the third argument to the setvbuf function.

The type **fpos_t** is defined in <stdio.h> for use with fgetpos and fsetpos.

<stdlib.h>

The <stdlib.h> include file declares the following functions:

abort	calloc	lldiv	realloc	system
abs	div	malloc	srand	wctomb
atexit	exit	mblen	strtod	wcstombs
atof	free	mbstowcs	strtol	
atoi	getenv	mbtowc	strtoll	
atol	labs	qsort	strtoul	
atoll	llabs	rand	strtoull	
bsearch	ldiv			

In extended mode, <stdlib.h> also defines the following standard extensions:

Include Files

_alloca	_dump_allocated_delta	_itoa	_rotl
_atold	_ecvt	_loadmod	_rotr
_beginthread	_enable	_lrotl	rpmatch
_crotl	_endthread	_llrotl	_searchenv
_crotr	_exit	_lrotr	_splitpath
csid	_fcvt	_llrotr	_srotl
_debug_calloc	_freemod	_ltoa	_srotr
_debug_free	_fullpath	_makepath	strtold
_debug_heapmin	_gcvt	max	swab
_disable	_getTIBvalue	min	_ultoa
_debug_malloc	_heapmin	_msize	_ulltoa
_debug_realloc	_heap_check	_onexit	
_dump_allocated	_interrupt	putenv	

Note: To use the debug memory management functions (_debug_calloc, _dump_allocated, and so on), you must compile with the debug memory (/Tm) option.

For a description of how these functions differ, see "Differentiating between Memory Management Functions" on page 23. For more information about the memory management functions, see Memory Management in the *Programming Guide*.

<stdlib.h> also contains definitions for the following macros:

NULL	The NULL pointer value. The value of NULL is 0 when in extended mode; otherwise, its value is ((void*)0). NULL is also defined in <locale.h>.
EXIT_SUCCESS	Expands to 0; used by the atexit function.
EXIT_FAILURE	Expands to 8; used by the atexit function.
RAND_MAX	Expands to an integer representing the largest number that the rand function can return.
MB_CUR_MAX	Expands to an integral expression representing the maximum number of bytes in a multibyte character for the current locale.

For more information on NULL and the types **size_t** and **wchar_t**, see "<stddef.h>" on page 773.

In extended mode, <stdlib.h> also defines the following global variables:

_doserrno	**_environ**
errno	**onexit_t**
_osmajor	**_osminor**
_osmode	

The variable **errno** is also defined in <stddef.h>. The variable **_doserrno** and the SAA definition of **errno** are provided in <errno.h>.

<stdlib.h> also defines the following **far** and **near** pointer macros to the corresponding standard library function:

_fcalloc	_ffree
_fmalloc	_fheapmin
_frealloc	_ncalloc
_nfree	_nmalloc
_nheapmin	_nrealloc

These macros are also defined in <malloc.h>.

<string.h>

The <string.h> include file declares the string manipulation functions:

memchr	strcat	strcspn	strncpy	strtok
memcmp	strchr	strerror	strpbrk	strxfrm
memcpy	strcmp	strlen	strrchr	
memmove	strcoll	strncat	strspn	
memset	strcpy	strncmp	strstr	

In extended mode, <string.h> also defines the following standard extensions:

memccpy	strdup	strlwr	strnset	strset
memicmp	_strerror	strnicmp	strrev	strupr
strcmpi	stricmp			

The mem*xxxx* functions are also included in "<memory.h>" on page 771.

<string.h> also defines the macro NULL and the type **size_t**.

For more information on NULL and the type **size_t**, see "<stddef.h>" on page 773. NULL is also defined in <locale.h>.

Include Files

string.h also defines the following **far** and **near** pointer macros to the corresponding standard library function:

_fmemccpy	_fmemset	_fstrdup	_fstrncpy	_fstrset
_fmemchr	_fstrcat	_fstricmp	_fstrnicmp	_fstrspn
_fmemcmp	_fstrchr	_fstrlen	_fstrnset	_fstrstr
_fmemcpy	_fstrcmp	_fstrlwr	_fstrpbrk	_fstrtok
_fmemicmp	_fstrcpy	_fstrncat	_fstrrchr	_fstrupr
_fmemmove	_fstrcspn	_fstrncmp	_fstrrev	_nstrdup

These macros are also defined in <memory.h>.

<sys\stat.h>

The <sys\stat.h> include file defines the low-level input/output functions fstat and stat. It also defines the structure **stat** and the following types:

dev_t **ino_t**
off_t **time_t**

These types are also defined in <sys\types.h>. Other low-level I/O functions are defined in <io.h>.

<sys\timeb.h>

The <sys\timeb.h> include file defines the _ftime function and also defines the type **time_t** and the structure **timeb**.

<sys\types.h>

The <sys\types.h> include file defines the following types:

ino_t **time_t**
dev_t **off_t**

These types are also defined in <sys\stat.h>.

<sys\utime.h>

The <sys\utime.h> include file defines the utime function, the structure **utimbuf**, and the type **time_t**.

<time.h>

The <time.h> include file declares the time and date functions:

asctime	ctime	gmtime	mktime	strptime
clock	difftime	localtime	strftime	time

In extended mode, <time.h> also defines the standard extensions _strdate, _strtime, and tzset.

<time.h> also provides:

A structure **tm** containing the components of a calendar time. See "gmtime — Convert Time" on page 289 for a list of the members of the **tm** structure.

A macro CLOCKS_PER_SEC equal to the number per second of the value returned by the clock function.

Types **clock_t**, **time_t**, and **size_t**.

The NULL pointer value.

For more information on NULL and the type **size_t**, see "<stddef.h>" on page 773.

<time.h> also defines the global variables **_daylight**, **_timezone**, and **_tzname**.

<umalloc.h>

The <umalloc.h> include file defines the memory allocation functions that are heap-specific:

_debug_ucalloc	_ucalloc	_udump_allocated	_uheapset
_debug_uheapmin	_uclose	_udump_allocated_delta	_uheap_walk
_debug_umalloc	_ucreate	_uheapchk	_umalloc
_mheap	_udefault	_uheap_check	_ustats
_uaddmem	_udestroy	_uheapmin	_uopen

<umalloc.h> also defines macros used by the heap-specific functions.

Definitions for the non-heap-specific memory management functions are located in <stdlib.h> and in <malloc.h>.

Note: To use the debug versions of memory management functions, specify the debug memory (/Tm) compiler option.

<variant.h>

The <variant.h> include file declares the getsyntx function and for the **struct variant** type:

```
struct variant {
        char *codeset;                  /* code set of the current locale */
        char    backslash;              /* encoding of \    */
        char    right_bracket;          /* encoding of ]    */
        char    left_bracket;           /* encoding of [    */
        char    right_brace;            /* encoding of }    */
        char    left_brace;             /* encoding of {    */
        char    circumflex;             /* encoding of      */
        char    tilde;                  /* encoding of ~    */
        char   exclamation_mark;        /*  encoding  of  !   */
        char    number_sign;            /* encoding of #    */
        char    vertical_line;          /* encoding of |    */
        char    dollar_sign;            /* encoding of $    */
        char    commercial_at;          /* encoding of @    */
        char    grave_accent;           /* encoding of `    */
}
```

For more information about the effect of locale, see "setlocale — Set Locale" on page 499, "<locale.h>" on page 766, individual functions, and the introduction to locale in the *Programming Guide*.

<wchar.h>

The <wchar.h> include file declares the supported subset of the ISO/C Multibyte Support extensions defined in ISO/IEC 9899:1990/Amendment 1:1993(E) extensions. The following functions are declared in <wchar.h>:

fgetwc	putwc	wcscpy	wcsstr
fgetws	putwchar	wcscspn	wcstod
fputwc	swprintf	wcsftime	wcstok
fputws	swscanf	wcslen	wcstol
getwc	ungetwc	wcsncat	wcstoul
getwchar	vswprintf	wcsncmp	wcswidth
getwmccoll	wcrtomb	wcsncpy	wcsxfrm
mbrlen	wcscat	wcspbrk	wctob
mbrtowc	wcschr	wcsrchr	wcwidth
mbsinit	wcscmp	wcsrtombs	
mbsrtowcs	wcscoll	wcsspn	

You do not need to include <stdio.h> and <stdarg.h> to use this include file.

<wchar.h> also defines the following types and constants:

mbstate_t Conversion state information needed when converting between sequences of multibyte characters and wide characters.

size_t **typedef** for the type of the value returned by **sizeof**.

wchar_t **typedef** for a wide character constant.

wint_t An integral type unchanged by integral promotions, that can hold any value corresponding to members of the extended character set, as well as WEOF (see below).

FILE The **FILE** structure type is defined in <stdio.h>. Stream functions use a pointer to the **FILE** type to get access to a given stream. The system uses the information in the **FILE** structure to maintain the stream. The C standard streams **stdin**, **stdout**, and **stderr** are also defined in <stdio.h>.

va_list This type is also defined in <stdio.h>.

NULL A pointer that never points to a data object.

WEOF Expands to a constant expression of type **wint_t**, whose value does not correspond to any member of the extended character set. It indicates EOF.

You can perform wide character input/output on the streams described in the ISO/IEC 9899:1990 standard, subclause 7.9.2. This standard expands the definition of a stream to include an *orientation* for both text and binary streams. After you have associated a stream with an external file, but before you have performed any operations on it, the stream does not have any orientation. Once you apply a wide character input or output function to a stream that does not have orientation, the stream becomes *wide-oriented*. Similarly, once you apply a byte input or output function to a stream without orientation, the stream becomes *byte-oriented*.

After you establish a stream's orientation, the only way to change it is to make a successful call to the freopen function, which removes a stream's orientation.

For more information about the effect of locale, see “setlocale — Set Locale” on page 499, “<locale.h>” on page 766, individual functions, and the introduction to locale in the *Programming Guide*.

Include Files

<wcstr.h>

The wcstr.h include file declares the multibyte functions:

wcscat	wcscpy	wcsncat	wcspbrk	wcswcs
wcschr	wcscspn	wcsncmp	wcsspn	
wcscmp	wcslen	wcsncpy	wcsrchr	

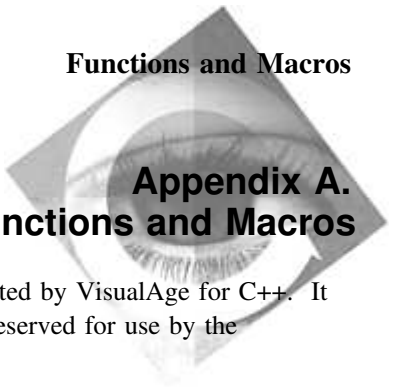
wcstr.h also defines the types **size_t**, NULL, and **wchar_t**.

<wctype.h>

The <wctype.h> file declares functions used to classify wide characters:

iswalnum	iswctype	iswprint	iswxdigit
iswalpha	iswdigit	iswpunct	towlower
iswblank	iswgraph	iswspace	towupper
iswcntrl	iswlower	iswupper	wctype

wctype.h also defines the types **wint_t** and **wctype_t**, and the macro WEOF, which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set and indicates EOF.



Appendix A. VisualAge for C++ Functions and Macros

This appendix lists the C functions and macros supported by VisualAge for C++. It also includes a list of the predefined macros that are reserved for use by the VisualAge for C++ product.

Functions and Macros

This section lists all the C functions and macros supported by VisualAge for C++ and classifies each according to the following table:

ANSI/ISO Defined in the ANSI/ISO 9899-1990[1992] C standard.

ANSI/ISO 93 Defined in the ISO/IEC 9899:1990/Amendment 1:1993(E).

POSIX Defined in the ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard.

XPG4 Defined in the X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4.

SAA Defined in the SAA Level 2 C standard.

Extension Extension to existing standards, specific to the VisualAge for C++ compiler.

Subsystem Included in the subsystem libraries.

If you are writing code to be ported to other standards-conforming systems, you can use the following table to find out which VisualAge for C++ functions are portable. You can also set the language level of your source code to ensure only portable functions are used:

To allow only functions defined by ANSI/ISO, set the language level to ANSI.
To allow SAA functions as well as ANSI/ISO functions, set the language level to SAA or SAAL2.
To allow all VisualAge for C++ functions, set the language level to Extended. This is the default.

Where a function is classified under a language standard and also as an extension, the function conforms to the standard indicated but has additional implementation-defined features. Such a function is considered to be an extension. If you set the language level to ANSI or SAA, you can still use the function but the additional features will not be available.

© Copyright IBM Corp. 1992, 1996

Table 9 (Page 1 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
abort	<stdlib.h> <process.h>	X		X	X	X		
abs	<stdlib.h>	X		X	X	X		X
access	<io.h>			X	X		X	X
acos	<math.h>	X		X	X	X		
_alloca	<stdlib.h> <malloc.h> <builtin.h>						X X	X
asctime	<time.h>	X		X	X	X		
asin	<math.h>	X		X	X	X		
assert	<assert.h>	X		X	X	X		
atan	<math.h>	X		X	X	X		
atan2	<math.h>	X		X	X	X		
atexit	<stdlib.h>	X			X	X		
atof	<stdlib.h>	X		X	X	X	X	X
atoi	<stdlib.h>	X		X	X	X		X
atol	<stdlib.h>	X		X	X	X		X
atoll	<stdlib.h>						X	X
_atold	<stdlib.h> <math.h>						X X	X
_beginthread	<stdlib.h> <process.h>						X X	

Table 9 (Page 2 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
bessel — _j0,_j1,_jn _y0,_y1,_yn	<math.h>					X		
bsearch	<stdlib.h> <search.h>	X		X	X	X	 X	X
_cabs	<math.h>						X	
calloc	<stdlib.h> <malloc.h>	X		X	X	X	 X	X
ceil	<math.h>	X		X	X	X		
_cgets	<conio.h>						X	
chdir	<direct.h>				X		X	
_chdrive	<direct.h>						X	
chmod	<io.h>				X		X	X
_chsize	<io.h>						X	X
_clear87	<float.h> <builtin.h>						X X	X
clearerr	<stdio.h>	X		X	X	X		
clock	<time.h>	X			X	X		
close	<io.h>				X		X	X
_control87	<float.h> <builtin.h>						X X	X
cos	<math.h>	X		X	X	X		
cosh	<math.h>	X		X	X	X		
_cprintf	<conio.h>						X	

Table 9 (Page 3 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_cputs	<conio.h>						X	
creat	<io.h>				X		X	X
_crotl	<stdlib.h> <builtin.h>						X X	
_crotr	<stdlib.h> <builtin.h>						X X	
_cscanf	<conio.h>						X	
csid	<stdlib.h>						X	
ctime	<time.h>	X		X	X	X		
_cwait	<process.h>						X	
__cxchg	<builtin.h>						X	
_debug_calloc	<stdlib.h> <malloc.h>						X X	X
_debug_free	<stdlib.h> <malloc.h>						X X	X
_debug_heapmin	<stdlib.h> <malloc.h>						X X	X
_debug_malloc	<stdlib.h> <malloc.h>						X X	X
_debug_realloc	<stdlib.h> <malloc.h>						X X	X
_debug_ucalloc	<umalloc.h>						X	X
_debug_uheapmin	<umalloc.h>						X	X

Table 9 (Page 4 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
_debug_umalloc	<umalloc.h>						X	X
difftime	<time.h>	X			X	X		
_disable	<stdlib.h> <builtin.h>						X X	
div	<stdlib.h>	X			X	X		X
_dump_allocated	<stdlib.h> <malloc.h>						X X	X
_dump_allocated_delta	<stdlib.h> <malloc.h>						X X	X
dup	<io.h>				X		X	X
dup2	<io.h>				X		X	X
_ecvt	<stdlib.h>						X	
_enable	<stdlib.h> <builtin.h>						X X	
_endthread	<stdlib.h> <process.h>						X X	
__eof	<io.h>						X	X
erf	<math.h>			X		X		
erfc	<math.h>				X	X		
execl	<process.h>				X		X	
execle	<process.h>				X		X	
execlp	<process.h>				X		X	
_execlpe	<process.h>						X	

Table 9 (Page 5 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
execv	<process.h>				X		X	
execve	<process.h>				X		X	
execvp	<process.h>				X		X	
_execvpe	<process.h>						X	
exit	<stdlib.h> <process.h>	X		X	X	X	 X	X
_exit	<stdlib.h> <process.h>			X			X X	
exp	<math.h>	X		X	X	X		
fabs	<math.h>	X		X		X		
_facos	<builtin.h>						X	
_fasin	<builtin.h>						X	
fclose	<stdio.h>	X		X	X	X		
_fcloseall	<stdio.h>						X	
_fcos	<builtin.h>						X	
_fcossin	<builtin.h>						X	
_fcvt	<stdlib.h>						X	
fdopen	<stdio.h>				X		X	
feof	<stdio.h>	X		X	X	X		
ferror	<stdio.h>	X		X	X	X		
fflush	<stdio.h>	X		X	X	X		
fgetc	<stdio.h>	X		X	X	X		

Table 9 (Page 6 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_fgetchar	<stdio.h>						X	
fgetpos	<stdio.h>	X			X	X		
fgets	<stdio.h>	X		X	X	X		
fgetwc	<wchar.h>		X					
fgetws	<wchar.h>		X X		X			
_filelength	<io.h>						X	
fileno	<stdio.h>				X		X	
floor	<math.h>	X		X	X	X		
_flushall	<stdio.h>						X	
fmod	<math.h>	X		X	X	X		
fopen	<stdio.h>	X		X	X	X		
_fpatan	<builtin.h>						X	
_fpreset	<float.h>						X	
fprintf	<stdio.h>	X		X	X	X	X	
_fptan	<builtin.h>						X	
fputc	<stdio.h>	X		X	X	X		
_fputchar	<stdio.h>						X	
fputs	<stdio.h>	X		X	X	X		
fputwc	<wchar.h>		X X		X			

Table 9 (Page 7 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
fputws	<wchar.h>		X X		X			
fread	<stdio.h>	X		X	X	X		
free	<stdlib.h> <malloc.h>	X		X	X	X	 X	X
_freemod	<stdlib.h>						X	
freopen	<stdio.h>	X		X	X	X		
frexp	<math.h>	X		X	X	X		
fscanf	<stdio.h>	X		X	X	X	X	
fseek	<stdio.h>	X		X	X	X		
fsetpos	<stdio.h>	X			X	X		
_fsin	<builtin.h>						X	
_fsincos	<builtin.h>						X	
_fsqrt	<builtin.h>						X	
fstat types for fstat	<sys\stat.h> <sys\types.h>			X	X		X	
ftell	<stdio.h>	X		X	X	X		
_ftime types for _ftime	<sys\timeb.h> <sys\types.h>						X	
_fullpath	<stdlib.h>						X	
fwrite	<stdio.h>	X		X	X	X		
_fyl2x	<builtin.h>						X	

Table 9 (Page 8 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
_fyl2xp1	<builtin.h>						X	
_f2xm1	<builtin.h>						X	
gamma	<math.h>					X		
_gcvt	<stdlib.h>						X	
getc	<stdio.h>	X		X	X	X		
_getch	<conio.h>						X	
getchar	<stdio.h>	X		X	X	X		
_getche	<conio.h>						X	
_getcwd	<direct.h>			X			X	
_getdcwd	<direct.h>						X	
_getdrive	<direct.h>						X	
getenv	<stdlib.h>	X		X	X	X		
getpid	<process.h>				X		X	
gets	<stdio.h>	X		X	X	X		
getsyntx	<variant.h>		X		X			
getwc	<wchar.h.>		X		X			
getwchar	<wchar.h>	X			X	X		
gmtime	<time.h>	X		X	X	X		
_heap_check	<stdlib.h> <malloc.h>						X X	X
_heapchk	<malloc.h>						X	X

Table 9 (Page 9 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_heapmin	<stdlib.h> <malloc.h>						X X	X
_heapset	<malloc.h>						X	X
_heap_walk	<malloc.h>						X	X
hypot	<math.h>				X	X		
iconv	<iconv.h>				X			
iconv_close	<iconv.h>				X			
iconv_open	<iconv.h>				X			
_inp	<conio.h> <builtin.h>						X X	
_inpd	<conio.h> <builtin.h>						X X	
_inpw	<conio.h> <builtin.h>						X X	
_interrupt	<stdlib.h> <builtin.h>						X X	
isalnum	<ctype.h>	X		X	X	X		
isalpha	<ctype.h>	X		X	X	X		
isascii	<ctype.h>				X		X	
isatty	<io.h>				X		X	X
isblank	<ctype.h>						X	
iscntrl	<ctype.h>	X		X	X	X		
_iscsym	<ctype.h>						X	

Table 9 (Page 10 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_iscsymf	<ctype.h>						X	
isdigit	<ctype.h>	X		X	X	X		
isgraph	<ctype.h>	X		X	X	X		
islower	<ctype.h>	X		X	X	X		
isprint	<ctype.h>	X		X	X	X		
ispunct	<ctype.h>	X		X	X	X		
isspace	<ctype.h>	X		X	X	X		
isupper	<ctype.h>	X		X	X	X		
iswalnum	<wctype.h>		X	X	X			
iswalpha	<wctype.h>		X	X	X			
iswblank	<wctype.h>						X	
iswcntrl	<wctype.h>		X	X	X			
iswctype	<wctype.h>		X	X	X			
iswdigit	<wctype.h>		X	X	X			
iswgraph	<wctype.h>		X	X	X			
iswlower	<wctype.h>		X	X	X			
iswprint	<wctype.h>		X	X	X			
iswpunct	<wctype.h>		X	X	X			
iswspace	<wctype.h>		X	X	X			
iswupper	<wctype.h>		X	X	X			
iswxdigit	<wctype.h>		X	X	X			

Table 9 (Page 11 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
isxdigit	<ctype.h>	X		X	X	X		
_itoa	<stdlib.h>						X	X
_kbhit	<conio.h>						X	
labs	<stdlib.h>	X				X		X
ldexp	<math.h>	X		X		X		
ldiv	<stdlib.h>	X				X		X
lfind	<search.h>				X		X	
llabs	<stdlib.h>					X	X	
lldiv	<stdlib.h>					X	X	
_llrotl	<stdlib.h> <builtin.h>					X		
_llrotr	<stdlib.h> <builtin.h>					X		
_loadmod	<stdlib.h>						X	
localdtconv	<locale.h>						X	
localeconv	<locale.h>	X			X	X		
localtime	<time.h>	X		X	X	X		
log	<math.h>	X		X	X	X		
log10	<math.h>	X		X	X	X		
longjmp	<setjmp.h>	X		X		X	X	X
_lrotl	<stdlib.h> <builtin.h>						X X	

Table 9 (Page 12 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_lrotr	<stdlib.h> <builtin.h>						X X	
lsearch	<search.h>				X		X	
lseek	<io.h>					X	X	X
_ltoa	<stdlib.h>						X	X
__lxchg	<builtin.h>						X	
_makepath	<stdlib.h>						X	
malloc	<stdlib.h> <malloc.h>	X		X	X	X	 X	X
_matherr	<math.h>						X	
max	<stdlib.h>						X	
mblen	<stdlib.h>	X			X	X		
mbrlen	<wchar.h>		X					
mbrtowc	<wchar.h>		X					
mbsinit	<wchar.h>						X	
mbsrtowcs	<wchar.h>		X					
mbstowcs	<stdlib.h>	X			X	X		
mbtowc	<stdlib.h>	X			X	X		
memccpy	<string.h> <memory.h>				X		X X	
memchr	<string.h> <memory.h>	X			X	X	 X	X

Table 9 (Page 13 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
memcmp	<string.h> <memory.h>	X			X	X	 X	X
memcpy	<string.h> <memory.h>	X			X	X	 X	X
memicmp	<string.h> <memory.h>						X X	
memmove	<string.h> <memory.h>	X			X	X	 X	X
memset	<string.h> <memory.h>	X			X	X	 X	X
_mheap	<umalloc.h>						X	X
min	<stdlib.h>						X	
mkdir	<direct.h>				X		X	
mktime	<time.h>	X		X	X	X		
modf	<math.h>	X		X	X	X		
_msize	<stdlib.h> <malloc.h>						X X	X
nl_langinfo types for nl_langinfo	<langinfo.h <nl_types.h>				X			
_onexit	<stdlib.h>						X	
open constants for open	<io.h> <fcntl.h>				X		X	X
_outp	<conio.h> <builtin.h>						X X	

Table 9 (Page 14 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
_outpd	<conio.h> <builtin.h>						X X	
_outpw	<conio.h> <builtin.h>						X X	
perror	<stdio.h>	X		X	X	X		
pow	<math.h>	X		X	X	X		
printf	<stdio.h>	X		X	X	X	X	X
putc	<stdio.h>	X		X	X	X		
_putch	<conio.h>						X	
putchar	<stdio.h>	X		X	X	X		
putenv	<stdlib.h>				X		X	
puts	<stdio.h>	X		X	X	X		
putwc	<wchar.h>		X X		X X			
putwchar	<wchar.h>		X		X			
qsort	<stdlib.h> <search.h>	X		X	X	X	 X	X
raise	<signal.h>	X			X	X		
rand	<stdlib.h>	X		X		X		
read	<io.h>				X		X	X
realloc	<stdlib.h> <malloc.h>	X		X	X	X	 X	X
regcomp	<regex.h>			X	X			

Table 9 (Page 15 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
regerror	<regex.h>			X	X			
regexec	<regex.h>			X	X			
regfree	<regex.h>			X	X			
remove	<stdio.h> <io.h>	X		X	X	X		
rename	<stdio.h> <io.h>	X		X	X	X		
rewind	<stdio.h>	X		X	X	X		
rmdir	<direct.h>				X		X	
_rmtmp	<stdio.h>						X	
_rotl	<stdlib.h> <builtin.h>						X X	
_rotr	<stdlib.h> <builtin.h>						X X	
rpmatch	<stdlib.h>			X			X	
scanf	<stdio.h>	X		X	X	X	X	
_searchenv	<stdlib.h>						X	
setbuf	<stdio.h>	X		X	X	X		
_set_crt_msg_handle	<stdio.h>						X	X
setjmp	<setjmp.h>	X		X		X	X	X
setlocale	<locale.h>	X		X	X	X		
_setmode types for _setmode	<io.h> <fcntl.h>						X	X

Table 9 (Page 16 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
setvbuf	<stdio.h>	X			X	X		
signal	<signal.h>	X			X	X	X	
sin	<math.h>	X		X	X	X		
sinh	<math.h>	X		X	X	X		
_sopen constants for _sopen	<io.h> <fcntl.h> <share.h>						X	X
_spawnl	<process.h>						X	
_spawnle	<process.h>						X	
_spawnlp	<process.h>						X	
_spawnlpe	<process.h>						X	
_spawnv	<process.h>						X	
_spawnve	<process.h>						X	
_spawnvp	<process.h>						X	
_spawnvpe	<process.h>						X	
_splitpath	<stdlib.h>						X	
sprintf	<stdio.h>	X		X	X	X	X	X
sqrt	<math.h>	X		X	X	X		
srand	<stdlib.h>	X		X	X	X		
_srotl	<stdlib.h> <builtin.h>						X X	
_srotr	<stdlib.h> <builtin.h>						X X	

Table 9 (Page 17 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
sscanf	<stdio.h>	X		X		X	X	X
stat types for stat	<sys\stat.h> <sys\types.h>				X		X	
_status87	<float.h> <builtin.h>						X X	X
strcat	<string.h>	X		X	X	X		X
strchr	<string.h>	X		X	X	X		X
strcmp	<string.h>	X		X	X	X		X
strcmpi	<string.h>						X	
strcoll	<string.h>	X			X	X		
strcpy	<string.h>	X		X		X	X	X
strcspn	<string.h>	X		X	X	X		X
_strdate	<time.h>						X	
strdup	<string.h>				X		X	X
strerror	<string.h>	X				X		
_strerror	<string.h>						X	
strfmon	<monetary.h>				X			
strftime	<time.h>	X		X	X	X		
stricmp	<string.h>						X	
strlen	<string.h>	X		X	X	X		X
strlwr	<string.h>						X	
strncat	<string.h>	X		X	X	X		X

Table 9 (Page 18 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
strncmp	<string.h>	X		X	X	X		X
strncpy	<string.h>	X		X	X	X		X
strnicmp	<string.h>						X	
strnset	<string.h>						X	
strpbrk	<string.h>	X		X	X	X		X
strptime	<time.h>				X			
strrchr	<string.h>	X		X	X	X		X
strrev	<string.h>						X	
strset	<string.h>						X	
strspn	<string.h>	X		X	X	X		X
strstr	<string.h>	X		X	X	X		X
_strtime	<time.h>						X	
strtod	<stdlib.h>	X			X	X	X	
strtok	<string.h>	X		X	X	X		
strtol	<stdlib.h>	X			X	X		X
strtoll	<stdlib.h>						X	X
strtold	<stdlib.h>						X	
strtoul	<stdlib.h>	X			X	X		X
strtoull	<stdlib.h>						X	X
strupr	<string.h>						X	
strxfrm	<string.h>	X			X	X		

Table 9 (Page 19 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
swab	<stdlib.h>				X		X	
swprintf	<wchar.h>		X					
swscanf	<wchar.h>		X					
__sxchg	<builtin.h>						X	
system	<stdlib.h> <process.h>	X		X		X	X X	
tan	<math.h>	X		X	X	X		
tanh	<math.h>	X		X	X	X		
_tcalloc	<stdlib.h> <malloc.h>						X X	
_tdump_allocated	<stdlib.h> <malloc.h>						X X	
_tdump_allocated_delta	<stdlib.h> <malloc.h>						X X	
_tell	<io.h>						X	X
tempnam	<stdio.h>				X		X	
_tfree	<stdlib.h> <malloc.h>						X X	
_theap_check	<stdlib.h> <malloc.y>						X X	
_theapmin	<stdlib.h> <malloc.h>						X X	
_threadstore	<stdlib.h>						X	

Table 9 (Page 20 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
time	<time.h>	X		X	X	X		
_tmalloc	<stdlib.h> <malloc.h>						X X	
tmpfile	<stdio.h>	X		X	X	X		
tmpnam	<stdio.h>	X		X	X	X		
_toascii	<ctype.h>						X	
tolower	<ctype.h>	X		X	X	X		
_tolower	<ctype.h>						X	
toupper	<ctype.h>	X		X	X	X		
_toupper	<ctype.h>						X	
towlower	<wctype.h>		X	X				
towupper	<wctype.h>		X	X				
_trealloc	<stdlib.h> <malloc.h>						X X	
tzset	<time.h>			X	X		X	
_uaddmem	<umalloc.h>						X	X
_ucalloc	<umalloc.h>						X	X
_uclose	<umalloc.h>						X	X
_ucreate	<umalloc.h>						X	X
_udefault	<umalloc.h>						X	X
_udestroy	<umalloc.h>						X	X
_udump_allocated	<umalloc.h>						X	X

Table 9 (Page 21 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
_udump_allocated_delta	<umalloc.h>						X	X
_uheap_check	<umalloc.h>						X	X
_uheapchk	<umalloc.h>						X	X
_uheapmin	<umalloc.h>						X	X
_uheapset	<umalloc.h>						X	X
_uheap_walk	<umalloc.h>						X	X
_ultoa	<stdlib.h>						X	X
_ulltoa	<stdlib.h>						X	X
_umalloc	<umalloc.h>						X	X
umask	<io.h>				X		X	X
ungetc	<stdio.h>	X		X	X	X		
_ungetch	<conio.h>						X	
ungetwc	<wchar.h>				X X			
unlink	<stdio.h> <io.h>				X		X	
_uopen	<umalloc.h>						X	X
_ustats	<umalloc.h>						X	X
utime types for utime	<sys\utime.h> <sys\types.h>				X		X	
va_arg	<stdarg.h>	X				X		X
va_end	<stdarg.h>	X				X		X

Table 9 (Page 22 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
va_start	<stdarg.h>	X				X		X
vfprintf	<stdarg.h> <stdio.h>	X X			X X	X X	X X	X X
vprintf	<stdarg.h> <stdio.h>	X X			X X	X X	X X	X X
vsprintf	<stdarg.h> <stdio.h>	X X			X X	X X	X X	X X
vswprintf	<wchar.h>		X					
wait	<process.h>				X		X	
wcrtomb	<wchar.h>		X					
wcscat	<wcstr.h> <wchar.h>				X	X		
wcschr	<wcstr.h> <wchar.h>				X	X		
wcscmp	<wcstr.h> <wchar.h>				X	X		
wcscoll	<wchar.h>		X		X			
wcscpy	<wcstr.h> <wchar.h>				X	X		
wcscspn	<wcstr.h> <wchar.h>				X	X		
wcsftime	<wchar.h>		X		X			
wcsid	<stdlib.h>						X	
wcslen	<wcstr.h> <wchar.h>				X	X		

Table 9 (Page 23 of 24). Functions Included in VisualAge for C++

Function/Macro	#include file	ANSI/ ISO	ANSI/ ISO 93	POSIX	XPG4	SAA	Extension	Subsystem
wcsncat	<wcstr.h> <wchar.h>				X	X		
wcsncmp	<wcstr.h> <wchar.h>				X	X		
wcsncpy	<wcstr.h> <wchar.h>				X	X		
wcspbrk	<wcstr.h> <wchar.h>				X	X		
wcsrchr	<wcstr.h> <wchar.h>				X	X		
wcsrtombs	<wchar.h>		X					
wcsspn	<wcstr.h> <wchar.h>				X	X		
wcsstr	<wchar.h>		X					
wcstod	<wchar.h>		X		X			
wcstok	<wchar.h>		X		X			
wcstol	<wchar.h>		X		X			
wcstombs	<stdlib.h>	X			X	X		
wcstoul	<wchar.h>	X			X	X		
wcswcs	<wcstr.h>				X	X		
wcswidth	<wchar.h>				X			
wcsxfrm	<wchar.h>		X		X			
wctob	<stdio.h> <wchar.h>		X X					

Table 9 (Page 24 of 24). Functions Included in VisualAge for C++

Function/Macro	#include **file**	**ANSI/ ISO**	**ANSI/ ISO 93**	**POSIX**	**XPG4**	**SAA**	**Extension**	**Subsystem**
wctomb	<stdlib.h>	X			X	X		
wctype	<wctype.h>		X		X			
wcwidth	<wchar.h>				X			
write	<io.h>				X		X	X

Predefined Macros

The macros identified in this section are provided to allow customers to write programs that use VisualAge for C++ services. Only those macros identified in this section should be used to request or receive VisualAge for C++ services.

VisualAge for C++ compiler provides both the SAA predefined macros and a number of macros specific to VisualAge for C++ product.

SAA Macros

Macro	Description
__LINE__	Represents the current source line number.
__FILE__	Indicates the name of the source file
__DATE__	Indicates the date when the source file was compiled.
__TIME__	Indicates the time when the source file was compiled.
__TIMESTAMP__	Indicates the date and time when the file was last modified.
__STDC__	Set to the integer 1. Indicates the compiler complies with ANSI C standards. This macro is defined for C programs only.
__ANSI__	Indicates only language constructs that conform to ANSI C standards are allowed. Defined using the **#pragma langlvl(ansi)** directive or /Sa compiler option.
__SAA__	Indicates only language constructs that conform to the most recent level of SAA C standards are allowed. Defined using the **#pragma langlvl(saa)** directive or /S2 compiler option. This macro is defined for C programs only.
__SAA_L2__	Indicates only language constructs that conform to SAA Level 2 C standards are allowed.. Defined using the **#pragma langlvl(saal2)** directive or /S2 compiler option. This macro is defined for C programs only.
__EXTENDED__	Indicates additional language constructs defined by the implementation are allowed. Under the VisualAge for C++ compiler, all language constructs are allowed. Defined using the **#pragma langlvl(extended)** directive or /Se compiler option.

VisualAge for C++ Macros

Macro	Description
_CHAR_UNSIGNED	Indicates default character type is unsigned. Defined when the **#pragma chars(unsigned)** directive is in effect, or when the /J+ compiler option is set.
_CHAR_SIGNED	Indicates default character type is signed. Defined when the **#pragma chars(signed)** directive is in effect, or when the /J- compiler option is set.

__COMPAT__ Indicates language constructs compatible with earlier versions of the C++ language are allowed. Defined using the **#pragma langlvl(compat)** directive or /Sc compiler option. This macro is defined for C++ programs only.

__cplusplus Set to the integer 1. Indicates the product is a C++ compiler. This macro is defined for C++ programs only.

__DBCS__ Indicates DBCS support is enabled. Defined using the /Sn compiler option.

__DDNAMES__ Indicates ddnames are supported. Defined using the /Sh compiler option.

__DEBUG_ALLOC__ Maps memory management functions to their debug versions. Defined using the /Tm compiler option.

__DLL__ **and** _DLL Indicates code for a DLL is being compiled. Defined using the /Ge- compiler option.

_FP_INLINE_ Inlines the trigonometric functions (cos, sin, and so on).

__FUNCTION__ Indicates the name of the function currently being compiled. For C++ programs, expands to the actual function prototype.

__HHW_INTEL__ Indicates that the host hardware is an Intel** processor.

__HOS_WIN__ Indicates that the host operating system is Windows.

__IBMC__ Indicates the version number of the VisualAge C compiler.

__IBMCPP__ Indicates the version number of the VisualAge C++ compiler.

__IMPORTLIB__ Indicates that dynamic linking is used. Defined using the /Gd option.

_M_I386 Indicates code is being compiled for a 386 chip or higher.

_M_IX86 Indicates code is being compiled for a 386 chip or higher. Defined using the /G3, /G4, and /G5 compiler options.

__MULTI__ **and** _MT Indicates multithread code is being generated. Defined using the /Gm compiler option.

__NO_DEFAULT_LIBS__ Indicates that information about default libraries will not be included in object files. Defined using the /Gn option.

__WINDOWS__ Set to the integer 1. Indicates the product is a Windows compiler.

__SOM_ENABLED__ Indicates that native SOM is supported.

__SPC__ Indicates the subsystem libraries are being used. Defined using the /Rn compiler option.

__TEMPINC__ Indicates the template-implementation file method of resolving template functions is being used. Defined using the /Ft compiler option.

_STDCALL_SUPPORTED Indicates that the stdcall calling convention is supported on Windows.

__THW_INTEL__ Indicates that the target hardware is an Intel processor.

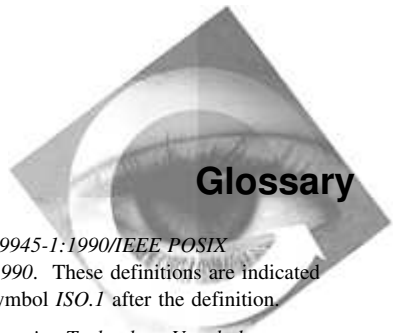
__TOS_WIN__ Indicates that the target operating system is Windows.

_WIN32 Indicates that the Win32 API set is supported.

Predefined Macros

__32BIT__ Set to the integer 1. Indicates the product is a 32-bit compiler.

The value of the __IBMC__ and __IBMCPP__ macros is 350. One of these two macros is always defined: when you compile C++ code, __IBMCPP__ is defined; when you compile C code, __IBMC__ is defined. The macros __WINDOWS__, _M_I386, and __32BIT__ are always defined also. The remaining macros, with the exception of __FUNCTION__, are defined when the corresponding **#pragma** directive or compiler option is used.



Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI* after the definition.

IBM Dictionary of Computing, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.

X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992. These definitions are indicated by the symbol *X/Open* after the definition.

ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990. These definitions are indicated by the symbol *ISO.1* after the definition.

The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abstraction (data). A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access mode. (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

American National Standards Institute. See *ANSI*.

ANSI (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI*.

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM*.

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components

© Copyright IBM Corp. 1992, 1996

used to perform specific types of user-oriented work on a computer. *IBM*.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

argument. (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

array. In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

array element. A data item in an array. *IBM*.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

Note: IBM has defined an extension to ASCII code (characters 128-255).

B

backslash. The character \. This character is named <backslash> in the portable character set.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

blank character. (1) A graphic representation of the space character. *ANSI*. (2) A character that represents an empty position in a graphic character string. *ISO Draft*. (3) One of the characters that belong to the *blank* character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open*.

block. (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with predefined. *IBM*.

C

C++ class library. See *class library*.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine. *IBM*.

cast. In the C and C++ languages, an expression that converts the type of the operand to a specified data type (the operator). *IBM*.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

character array. An array of type char. *X/Open*.

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open*.

character constant. (1) A constant with a character value. *IBM*. (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM*. (3) In some languages, a character enclosed in apostrophes. *IBM*.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*. (4) See also *portable character set*.

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class library. A collection of C++ classes.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI*. (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collection. (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

condition. A relational expression that can be evaluated to a value of either true or false. *IBM*.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with nonprinting character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion descriptor. A per-process unique value used to identify an open codeset conversion. *X/Open*.

coordinated universal time (UTC). Equivalent to Greenwich Mean Time (GMT)

current working directory. (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open*. *ISO.1*. (2) The directory that is searched when a file name is entered with no indication of the directory that lists the file name. The operating system assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*.

D

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM*.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM*.

data type. The properties and internal representation that characterize data.

DBCS (double-byte character set). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

declaration. (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by new.

device. A computer peripheral or an object that appears to the application as such. *X/Open*. *ISO.1*.

directory. A type of file containing the names and controlling information for other files or other directories. *IBM*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open*.

dot. The file name consisting of a single dot character (.). *X/Open*. *ISO.1*.

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI*.

dump. To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters. *IBM*.

E-format. Floating-point format, consisting of a number in scientific notation. *IBM*.

element. The component of an array, subrange, enumeration, or set.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

epoch. The time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time. *X/Open*. *ISO.1*.

exception. (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1*.

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM*.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The

internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

extension. (1) An element or function not included in the standard language. (2) File name extension.

F

file mode. An object containing the *file mode bits* and file type of a file, as described in <sys/stat.h>. *X/Open*.

file mode bits. A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID). *X/Open*.

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

for statement. A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM*.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM*.

G

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by *coordinate universal time (UTC)*.

Greenwich Mean Time. See GMT.

H

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap. An unordered flat collection that allows duplicate elements.

I

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM*.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM*.

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class box is

previously defined, two instances of a class box could be instantiated with the declaration:

```
box box1, box2;
```

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

Synonymous with *I18N*.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

K

key access. A property that allows elements to be accessed by matching keys.

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

L

label. An identifier within or attached to a set of data elements. *ISO Draft*.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM*.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM*.

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1*.
(2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI*.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open*.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. *ISO-JTC1*. *ANSI*.

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

method. In the C++ language, a synonym for member function.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM*.

mode. A collection of attributes that specifies a file's type and its access permissions. *X/Open*. *ISO.1*.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1*. *ANSI*.

mutex. (1) In Windows, a flag that prevents threads from interacting with the 16-bit kernel when another thread is executing code there.

N

name. In the C++ language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, destructor name or qualified name.

NULL. In the C and C++ languages, a pointer that does not point to a data object. *IBM*.

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value 00, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open*.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

null value. A parameter position for which no value is specified. *IBM*.

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and member functions that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and member functions. Each object in the class is said to be an instance of the class. (3) In Visual Builder, an instance of an object class consisting of attributes, a data structure, and operational member functions. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and member functions as other instances of the object class, though it has unique values assigned to its attributes. (4) In Windows, any item that is or can be linked into another Windows application, such as a sound, graphic, piece of text, or portion of a spreadsheet. An object must be from an application that supports OLE. See object linking and embedding (OLE).

object linking and embedding (OLE). (1) An API that supports compound documents, cross-application macro control, and common object registration. OLE defines protocols for in-place editing, drag-and-drop data transfers, structured storage, custom controls, and more. (2) A data-sharing scheme that allows dissimilar applications to create single complex documents cooperatively. The documents can consist of material that a single application could not have created on its own.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That

portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

P

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

parent process. (1) The program that originates the creation of other processes by means of spawn or exec function calls. See also *child process*. (2) A process that creates other processes.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

period. The character (**.**). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM*.

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open*.

process. (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. *X/Open*. *ISO.1*.

process ID. The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open*. *ISO.1*.

public. Pertaining to a class member that is accessible to all functions.

R

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

regular expression. (1) A mechanism to select specific strings from a set of character strings.

(2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

root. The base directory in the operating system.

runtime library. A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

S

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

semaphore. A synchronization kernel object used for resource-counting. A semaphore offers a thread the ability to query the number of resources available. If one or more resources are available, the count of available resources is decremented.

sequence. A sequentially ordered flat collection.

signal. (1) A condition that may or may not be reported during program execution. For example, SIGFPE is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open. ISO.1*.

signal handler. A function to be called when the signal is reported.

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open*.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

standard error (stderr). An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input (stdin). (1) An input stream usually intended to be used for primary data input. *X/Open*. (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM*.

standard output (stdout). (1) An output stream usually intended to be used for primary data output. *X/Open*. (2) In the AIX operating system, the primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM*.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the fdopen or fopen functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open*.

stream buffer. A stream buffer is a buffer between the ultimate consumer, ultimate producer, and the I/O Stream Library functions

that format data. It is implemented in the I/O Stream Library by the streambuf class and the classes derived from streambuf.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

string constant. Zero or more characters enclosed in double quotation marks.

struct. An aggregate of elements, having arbitrary types.

structure. A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

T

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). *X/Open*.

this. A C++ keyword that identifies a special type of pointer in a member function, that references the class object with which the member function was invoked.

thread. (1) The smallest unit or path of execution within a process. *IBM*. (2) A piece of executing code. (3) In Windows, each thread is allocated its own stack from the owning process' 4-GB address space, and each one has its own set of processor registers, called the thread's context.

tilde. The character ~. This character is named <tilde> in the portable character set.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM*.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. *ISO-JTC1*.

type. The description of the data and the operations that can be performed on or by the data. See also *data type*.

type conversion. Synonym for *boundary alignment*.

type definition. A definition of a name for a data type. *IBM*.

type specifier. Used to indicate the data type of an object or function being declared.

U

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM*.

union. (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

V

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

W

while statement. A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

Win32. The name of a 32-bit application programming interface (API).

See also Win32 API.

Win32 API. (1) A set of Win32 functions that can be called from source code. (2) A 32-bit version of the 16-bit Windows 3.1 API (native to Windows NT).

See also Win32.

Win32s. A platform that the Win32 API is implemented on. (The s stands for subset.) It consists of a virtual-device driver and dynamic link libraries (DLLs) that add the Win32 API to the 16-bit Windows 3.n system. It includes structured exception handling and limited implementations of memory-mapped files.

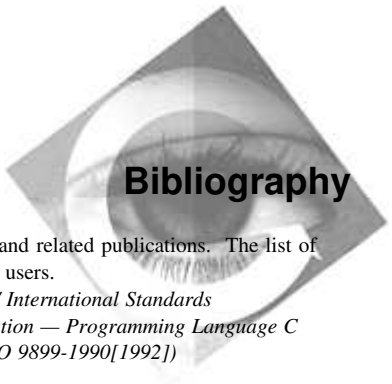
See also Win32 API and Windows 95.

Windows NT. A platform that the Win32 API is implemented on. It is a portable, high-end operating system, which can run several different types of applications simultaneously. It is the only Win32 platform for machine architectures based on processors other than the x86, and it supports multiple processors.

See also Win32 API.

Windows 95. (1) A 32-bit operating system that allows you to run 32-bit application. Windows 95 is a multitasking, multithreaded operating system that can control multiple programs at once. Each program can have multiple concurrent threads or independently executing subcomponents. (2) A platform that the Win32 API is implemented on. It supports image color matching, modems, and other services. It partially supports asynchronous file I/O, debugging, registry, security, and event-logging functions.

working directory. Synonym for *current working directory*.



Bibliography

This bibliography lists the publications that make up the IBM VisualAge for C++ library and related publications. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ users.

The IBM VisualAge for C++ Library

The following books are part of the IBM VisualAge for C++ library.

Installation Guide and Product Overview, S33H-5030

User's Guide, S33H-5031

Programming Guide, S33H-5032

Visual Builder User's Guide, S33H-5034

Visual Builder Parts Reference, S33H-5035

Building VisualAge for C++ Parts for Fun and Profit, S33H-5036

Open Class Library User's Guide, S33H-5033

Open Class Library Reference, S33H-5039

Language Reference, S33H-5037

C Library Reference, S33H-5038

SOM Programming Guide, S33H-5043

SOM Programming Reference, S33H-5044

C and C++ Related Publications

Portability Guide for IBM C, SC09-1405

American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

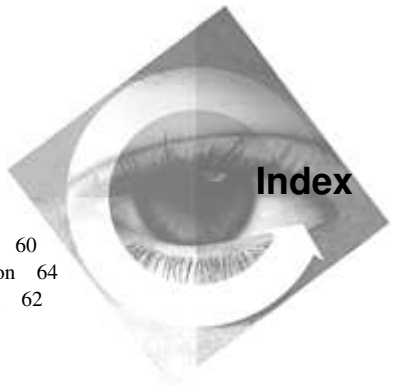
The Annotated C++ Reference Manual by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.

C++ Primer by Stanley B. Lippman, Addison-Wesley Publishing Company.

Object-Oriented Design with Applications by Grady Booch, Benjamin/Cummings.

Object-Oriented Programming Using SOM and DSOM by Christina Lau, Van Nostrand Reinhold.

© Copyright IBM Corp. 1992, 1996



Index

A

abnormal program termination 39
abort function 39
abs function 40
absolute value
  of complex number 73
  of floating-point number 182
  of integer 40
  of long integer 339
  of long long integer 344
_access function 41
access mode 194, 218
acos function 43
adding data to streams 195
_alloca function 45
allocating storage
  getting information about 154, 157, 656, 659
  reallocating 132
  temporarily 45
  with debug memory functions 124, 130
ANSI
  language level 783
appending 218, 242
appending data to files 195
arccosine 43
arcsine 49
arctangent 53
argument list functions 16
ASCII
  converting characters to 628
  testing for alphabetic or underscore 325
  testing for alphabetic, digit, or underscore 325
  testing for integer value 319
asctime function 47
asin function 49
assert function 51
assert.h include file 51, 761
assertions 51
assigning buffers
atan function 53
atan2 function 53
atexit function 54, 416
atof function 56
atoi function 58
atol function 60
_atold function 64
atoll function 62

B

_beginthread function 66
bessel functions 9, 69
binary files 219
binary search 71
bit mask
  fstat function 255
  stat function 532
  with _control87 function 94
blksize 219
block size 219
BookManager
bsearch function 71
buffers
  clearing 215
  comparing 400
bufsiz constant 774
built-in functions 45
builtin.h include file 761
byte swapping 608

C

_cabs function 73
calloc function 75
case mapping functions 21
case sensitivity
  comparing strings lowercase 562
  comparing strings without 540, 572
case, converting 628
ceil function 77
ceiling function 77
_cgets function 78
changing
  current drive 82
  default drive 80
  directory 80
  file handles 495
  file size 85
changing current file position 247

© Copyright IBM Corp. 1992, 1996

character testing functions 20
characters
converting
code set 302
date and time to wide string 719
multibyte to wide 386, 393
to floating-point 56, 58, 60, 62
to integer 58
to long double 64
to long integer 60, 62
wide to integer value 741
wide to multbyte 753
wide to multibyte 706
wide to single-byte 752
wide-character string to multibyte string 743
wide-character string to unsigned long value 745
printing to screen 98
reading from keyboard 272
setting 403
to upper case 628
_chdir function 80
_chdrive function 82
child processes
arguments for 176, 520
creating 519
environment 176
exit code 521
exit value 180
passing arguments from 116
passing arguments to 176, 520
passing environment to 176, 439
passing to child process 176
path name for 175
return status of 116
signal settings 176, 520
translation mode default 520
waiting for 116
child threads
See also threads
passing values to 66
_chmod function 83
_chsize function 85
classifying
clear error indicators 87
_clear87 function 88
clearerr 87
clearing
buffers 215
clearing *(continued)*
EOF 465
error indicator 465
clock function 90
CLOCKS_PER_SEC 90
_close function 92
closing
files 92
streams 187, 188
code set conversion 302
commands, passing within a program 613
complex number
absolute value of 73
concatenating strings 536, 566
conio.h include file 762
context-sensitive help
control word, floating-point 94
_control87 function 94
converting
characters
to lowercase 631
to uppercase 631
code set 302
converting character case 628
date and time to structure 576
date and time to wide character string 719
floating point to string 166, 192, 268
floating-point numbers to integers and fractions 412
from structure to string 47
integer to string 336
long integer to string 367
lower to uppercase 604
monetary value to string 553
multibyte character to wide-character 393
multibyte characters to wide characters 386
multibyte strings to wide-character strings 389, 391
string to formatted time 557
string to long double 64, 596
strings to floating-point values 56
strings to integer values 58
strings to long values 60, 62
unsigned long long to string 676
unsigned long to string 675
upper to lowercase 565
wide character case 632
wide character to multibyte character 706, 753
wide character to single-byte character 752

converting *(continued)*
wide-character string to double value 737
wide-character string to integer value 741
wide-character string to multibyte string 733, 743
wide-character string to unsigned long value 745
copying
bytes 395, 399, 402
strings 544, 549, 570
cos function 96
cosh function 97
_cprintf function 98
_cputs function 99
_creat function 100
creating
a temporary file 626, 627
directory 407
path name 374
_crotl function 102
_crotr function 102
_CRT_init function
_CRT_term function
_cscanf function 112
csid function 111
ctime function 114
__ctordtorInit function
__ctordtorTerm function
ctype.h include file 762
currency functions 10
current time
obtaining 259
_cwait function 116

D

data items 236
reading 236
writing 263
data type limits 766
date
correcting for local time 356
functions 10
representing as string 548
storing in a buffer 548
__DEBUG_ALLOC__ macro
_dump_allocated_delta function 157
_debug_calloc function 124, 134
_debug_free function 126
debugging memory management functions 16
_debug_heapmin function 128, 136
_debug_malloc function 130
_debug_realloc function 132
_debug_tcalloc function
_debug_tfree function
_debug_theapmin function
_debug_tmalloc function
_debug_trealloc function
_debug_ucalloc function
_debug_uheapmin function
_debug_umalloc function 138
default drive 80
deleting
directories 467
files 688
differential equations 9, 69
difftime function 140
direct.h include file 762
directory
changing 80
creating 407
current working 80
removing 467
renaming 464
directory management functions 6
_disable function 142
div function 143
_DLL_InitTerm function 144
dtconv structure 768
_dump_allocated function 154
_dup function 160
_dup2 function 163
duplicating a file handle 160, 163
dynamic memory management functions 16

E

echoing characters 272
_ecvt function 166
EDOM 763
_enable function 168
end of file, determining 171
end-of-file indicator 87, 197
end-of-stream
I/O 197
ending a program 39
ending threads 169
_endthread function 169

Enhanced editor (EPM)
environment table 279
environment variables 279
  modifying from within program 439
environment-affecting functions 18
EOF
  clearing 465
  macro 774
  resetting error indicator 87
__eof function 171
EPM
  See Enhanced editor (EPM)
ERANGE 763
erf function 173
erfc function 173
errno 763
errno variable 427
errno.h include file 763
error handling functions 5
error indicator 198
  clearing 465
error messages 427
  pointing to 551
  printing 551
exception masking 94
_exception_dllinit function
  refid termin.DLL environment 107
_exec functions
  _execlpe 175
  _execvpe 175
  execl 175
  execle 175
  execlp 175
  execv 175
  execve 175
  execvp 175
_exit function 179, 180
EXIT_FAILURE 179, 776
EXIT_SUCCESS 179, 776
exiting
  a process 180
  a program 179
exp function 181
extensions to standards
  library functions
    _access 41
    _alloca 45
    _atold 64
    _beginthread 66
    _cabs 73
    _cgets 78
extensions to standards *(continued)*
  library functions *(continued)*
    _chdir 80
    _chdrive 82
    _chmod 83
    _chsize 85
    _clear87 88
    _close 92
    _control87 94
    _cprintf function 98
    _cputs 99
    _creat 100
    _cscanf function 112
    _cwait 116
    _debug_calloc 124
    _debug_free 126
    _debug_heapmin 128
    _debug_malloc 130
    _debug_realloc 132
    _debug_tfree
    _debug_trealloc
    _disable 142
    _dump_allocated 154
    _dup 160
    _dup2 163
    _ecvt 166
    _enable 168
    _endthread 169
    __eof 171
    _execl 175
    _execle 175
    _execlp 175
    _execlpe 175
    _execv 175
    _execve 175
    _execvp 175
    _execvpe 175
    _exit 180
    _f2xm1 266
    _facos 183, 189
    _fasin 185
    _fcloseall 188
    _fcos
    _fcossin 190
    _fcvt 192
    _fdopen 194
    _fgetchar 202
    _filelength 211
    _fileno 213
    _flushall 215
    _fpatan 221

extensions to standards *(continued)*
library functions *(continued)*
_fpreset 222
_fptan 226
_fputchar 229
_freemod 240
_fsin 251
_fsincos 252
_fsqrt 254
_fstat 255
_ftime 259
_fullpath 261
_fyl2x 264
_fyl2xp1 265
_gcvt 268
_getch 272
_getche 272
_getcwd 274
_getdcwd 276
_getdrive 278
_getpid 280
_getTIBvalue
_heap_check 291
_heapmin 295
_inp 308
_inpd 310
_inpw 312
_interrupt 314
_isascii 319
_iscsym 325
_iscsymf 325
_itoa 336
_kbhit 337
_lfind 342
_llrotl 346
_llrotr 346
_loadmod 348
_lrotl 364
_lrotr 364
_lsearch 342
_lseek 365
_ltoa 367
_makepath 374
_matherr
max macro 381
memccpy 395
memicmp 400
min 406
_mkdir 407
_msize 413
_onexit 416

extensions to standards *(continued)*
library functions *(continued)*
_open 418
_outp 421
_outpd 423
_outpw 425
__parmdwords
_putch 438
_putenv 439
_read 450
_rmdir 467
_rmtmp 482
_rotl 483
_rotr 483
_searchenv 492
_set_crt_msg_handle 495
_setmode 505
signal 510
_sopen 516
_spawnl 519
_spawnle 519
_spawnlp 519
_spawnlpe 519
_spawnv 519
_spawnve 519
_spawnvp 519
_spawnvpe 519
_splitpath 523
_stat 532
_status87 534
_strdate 548
strdup 549
_strerror 551
stricmp 562
strlwr 565
strnicmp 572
_strtime 589
strtold 596
strupr 604
_swab 608
system 613
_tcalloc
_tell 619
_tempnam 621
_tfree
_threadstore 623
_tmalloc
_toascii 628
_tolower 628
_toupper 628
_trealloc

extensions to standards *(continued)*
library functions *(continued)*
_tzset 634
_ulltoa 676
_ultoa 675
_umask 679
_ungetch 683
_unlink 688
_utime 693
_wait
_write 759
predefined macros 808

F

_f2xm1 function 266
fabs function 182
_facos function 183, 189
_fasin function 185
_fclose function 187
_fcloseall function 188
fcntl.h 763
_fcos function
_fcossin function 190
_fcvt function 192
_fdopen function 194
feof function 197
ferror function 198
_fflush function 199
fgetc function 200
_fgetchar function 202
fgetpos function 203
_fgets function 205
fgetwc 207
fgetws function 209
file errors 87
file handles 160, 163
changing 495
file management functions 6
file modification time, setting 693
file name length 774
file names, temporary 775
file pointer, move 365
file positioning 203, 247, 249, 257
file status information 532
FILE type 775
_filelength function 211
_fileno function 213
files
appending to 195
files *(continued)*
changing permission mode 83
create and open 100
creating temporary 621
deleting 688
handle 213
include 761
maximum opened 774
name length 774
positioning 465
renaming 464
searching for 492
setting modification time 693
sharing 516
status information 532
temporary
See temporary files
unlinking 688
updating 194
float.h include file 763
floating point
control word
setting 94
converting
to string 166, 192, 268
exception masking 94
fast execution
status word
clearing 88
getting 534
unit, resetting 222
floor function 214
_flushall function 215
flushing buffers 199, 215
fmod function 217
fopen function 218
fopen, maximum simultaneous files 774
formatted I/O 224
_fpatan function 221
fpos_t 775
_fpreset function 222
fprintf function 224
_fptan function 226
fputc function 227
character to stdout 229
_fputchar function 229
fputs function 230
fputwc function 232
fputws function 234

fread function 236
free function 238
freeing
storage 126, 238
user DLLs 240
_freemod function 240
freopen function 242
frexp function 244
fscanf function 245
fseek function 247
fsetpos function 249
_fsin function function 251
_fsincos function 252
_fsqrt function 254
_fstat function 255
ftell function 257
_ftime function 259
_fullpath. function 261
function declarations 761
functions
See also extensions to standards
built-in 45
classification of 783
debug memory management
inlined 22
table of 784—807
fwrite function 263
_fyl2x function 264
_fyl2xp1 function 265

G

gamma function 267
_gcvt function 268
getc function 270
_getch function 272
getchar function 270
_getche function 272
_getcwd function 274
_getdcwd function 276
_getdrive function 278
getenv function 279
_getpid function 280
gets function 281
getsyntx function 283
_getTIBvalue function
getting
character property handle 755
current drive 278
current file position 257
getting *(continued)*
current working directory 274, 276
directory status information 532
file length 211
file position 203
file status information 532
floating-point status word 534
full path name 261
information about heap 691
information about memory 154, 157, 656, 659
pointer position 619
process identifier 280
wide character from stdin 287
getwc function 285
getwchar function 287
global variables
errno 427
P_tmpdir 621
gmtime function 289

H

handling interrupt signals 510
heap
checking storage in 291
getting information about 656, 659, 691
returning memory from 128, 295
heap checking functions 17
heap creation functions 18
_heap_check function 291
_heapmin function 295
HUGE 73, 770
HUGE_VAL 73, 770
hyperbolic
cosine 97
sine 515
tangent 618
hypot function 301
hypotenuse 301

I

I/O errors 87
I/O functions
low-level 14
stream 11
iconv function 302
iconv_close function 305

iconv_open function 306
iconv.h include file 763
include files
assert.h 761
builtin.h 761
conio.h 762
ctype.h 762
direct.h 762
errno.h 763
fcntl.h 763
float.h 763
iconv.h 763
io.h 764
limits.h 766
locale.h 766
malloc .h 769
math.h 770
memory.h 771
monetary.h 771
nl_types.h 771
process.h 772
regex.h 772
search.h 772
setjmp.h 773
share.h 773
signal.h 773
stdarg.h 773
stddef.h 773
stdio.h 774
stdlib.h 775
string.h 777
sys\stat.h 778
sys\timeb.h 778
sys\types.h 778
sys\utime.h 778
time.h 779
umalloc.h 779
wcstr.h 782
wctype.h 782
whcar.h 780
indicators, error 87
infinity values
description 27
in library functions
math functions 32, 34
printf family 30
scanf family 28
string conversion functions 31
macro constants for 27
initializing
constructors and destructors 103
DLL environment 103
storage
strings 570, 574
_inp function 308
_inpd function 310
input/output (I/O) stream
access mode 242
appending 218, 242
binary mode 242
buffering 493
changing current file position 247, 257
changing file position 465
closing 187
formatted I/O 245, 429, 486, 525, 530
opening 218
reading characters 200, 270
reading data items 236, 308, 310, 312
reading lines 205, 281
reopening 242
rewinding 465
text mode 242
translation mode 242
ungetting characters 681
updating 218, 242
writing characters 227, 436
writing data items 263, 421, 423, 425
writing lines 441
writing strings 230
_inpw function 312
integers
converting to strings 336, 367, 675, 676
pseudo-random 449
_interrupt function 314
interrupts
calling 314
disabling 142
enabling 168
io. h include file 764
isalnum function 315
isalpha function 315
_isascii function 319
_isatty function 321
isblank function 323
iscntrl function 315
_iscsym function 325
_iscsymf function 325
isdigit function 315

isgraph function 315
islower function 315
isprint function 315
ispunct function 315
isspace function 315
isupper function 315
iswalnum function 327
iswblank function 331
iswcntrl function 327
iswctype function 333
iswdigit function 327
iswgraph function 327
iswlower function 327
iswprint function 327
iswpunct function 327
iswspace function 327
iswupper function 327
iswxdigit function 327
isxdigit function 315
_itoa function 336

J

_j0,_j1,_jn (bessel functions) 69

K

_kbhit function 337
keyboard, writing characters to 683
keystrokes, checking for 337

L

labs function 339
ldexp function 340
ldiv function 341
length function 564
length in bytes, file 211
_lfind function 342
_LHUGE 770
_LHUGE_VAL 770
library functions
  See also functions, extensions to SAA
  classification of 783
  infinity and NaN values in 28
  intrinsic 22
  table of 784—807
library introduction 5
limits.h include file 766
lines
  reading 205
  writing 441
llabs function 344
lldiv function 345
_llrotl function 346
_llrotr function 346
loading DLLs 348
_loadmod function 348
local time
  copying into buffer 589
  difference from Universal Time 259
  file modification 693
  obtaining 259
local time corrections 356
localdtconv function 350
locale functions 18
locale.h include file 766
localeconv function 352
localtime function 356
locating storage 238
log function 358
log10 function 359
logic errors 51
logical record length 219
longjmp function 360
loss of significance errors 378
lrecl 219
_lrotl function 364
_lrotr function 364
_lsearch function 342
_lseek function 365
_ltoa function 367
__lxchg function 369

M

macros
  __DEBUG_ALLOC__
  for NaN and infinity values 27
  max 381
  min 406
  predefined 808
  __TILED__
_makepath function 374
malloc function 376
malloc.h include file 769
masking floating-point exceptions 94
math functions
  infinity and NaN values in 32, 34

math.h include file 770
mathematical functions 7
_matherr function 378
max function 381
maximum
  file name 774
  opened files 774
  temporary file name 775
MB_CUR_MAX 776
mblen function 382
mbrlen function 384
mbrtowc function 386
mbsinit function 388
mbsrtowcs function 389
mbstowcs function 391
mbtowc function 393
memccpy function 395
memchr function 396
memcmp function 397
memcpy function 399
memicmp function 400
memmove function 402
memory
  See also storage
  checking blocks of 291
  returning from heap 295
memory allocation functions 16
memory attribute
  freeing 126
  getting information about 154, 157, 656, 659
  returning from heap 128
memory management functions 16
memory object functions 18
memory.h include file 771
memset function 403
min macro 406
miscellaneous functions
  getenv 279
  longjmp 360
  perror 427
  rand 449
  setjmp 497
  srand 528
_mkdir function 407
mktime function 410
modf function 412
modification time, file 693
modifying environment variables 439
monetary functions 10
monetary.h include file 771
_msize function 413
multibyte conversions
  from wide character 706, 753
  from wide-character string 733, 743
multithread programs
  &I2@MULTITH.
    ending threads 169
  creating threads 66

N

NaN (not-a-number) values
  description 27
  in library functions
    math functions 32, 34
    printf family 30
    scanf family 28
    string conversion functions 31
  macro constants for 27
NDEBUG 51, 761
nl_langinfo function 414
  at end of program 416
nl_types.h include file 771
nonlocal goto 360, 497
NULL pointer 774, 775, 776

O

offsetof macro 774
_onexit function 416
open flag 418
_open function 418
OS/2
_outp function 421
_outpd function 423
_outpw function 425
overflow error 378

P

P_tmpdir 775
P_tmpdir variable 621
parent process
  delaying 116
  end of 175
  overlaying of 177
  passing envrionment to child 520
  suspension of 519
  waiting for exit of child 180

__parmdwords function
passing
  arguments 116
  commands within a program 613
  environment setting 176
  from child process 116
  to child process
    arguments 176
    environment settings 520
  values to child thread 66
path names
  creating 374
  decomposing 523
  getting 261
permission mask file 679
permission settings
  changing 83
  determining 41
  file 100
perror function 427
pointer position 619
pointers
portability
  publications 823
  standards to follow 1
pow function 428
predefined macros 808
printf function 429
printf functions
  infinity and NaN values in 30
printing
  characters to screen 98
  data items from stream 263
process control functions 5
process.h include file 772
processes
  child
    See child processes
  identifier 280
  spawning 519
pseudo-random integers 449
pseudorandom number functions 16
ptrdiff_t 773
publications
  related 823
pushing characters 681
putc function 436
_putch function 438, 450
putchar function 436
_putenv function 439
puts function 441
putwc function 442
putwchar function 444

Q

qsort function 446
quick sort 446

R

raise function 448
rand function 449
RAND_MAX 776
random access 247, 257
random number generator 449, 528
read operations 200
  character from **stdin** 200
read operations from keyboard 272
read to buffer, file 450
realloc function 452
reallocation 452
recfm 219
record format 219
redirection 242
regcomp function 455
regerror function 457
regex.h include file 772
regexec function 459
regfree function 462
regular expression functions 7
regular expressions
  compiling 455
related publications
  portability 823
  VisualAge for C++ 823
remove function 463
removing
  temporary files 482
rename function 464
renaming
  files 464
reopening streams 242
reserving
  storage
    _debug_calloc 124
    calloc 75
resetting floating-point unit 222

reversing strings 583
rewind function 465
rewinding
  a stream 465
_rmdir function 467
_rmtmp function 482
rotating bits 364
  characters 102
  unsigned integer value 483
  unsigned short values 529
_rotl function 483
_rotr function 483
rpmatch function 485

S

SAA functions
  absolute value
    abs 40
    fabs 182
    labs 339
    llabs 344
  character classification and conversion
    tolower 631
    toupper 631
  data conversion
    atof 56
    atoi 58
    atol 60
    atoll 62
  differential equations
    bessel 69
  exponential
    exp 181
    frexp 244
    ldexp 340
    log 358
    log10 359
    pow 428
  file handling
    remove 463
    rename 464
    tmpnam 627
  locale
    localeconv 352
  logarithmic
    log 358
    log10 359
  math
    acos 43
    asin 49
SAA functions *(continued)*
  math *(continued)*
    atan 53
    atan2 53
    bessel 69
    ceil 77
    cos 96
    cosh 97
    div 143
    erf 173
    erfc 173
    exp 181
    fabs 182
    floor 214
    fmod 217
    frexp 244
    gamma 267
    hypot 301
    ldexp 340
    ldiv 341
    lldiv 345
    log 358
    log10 359
    modf 412
    sin 514
    sinh 515
    sqrt 527
    tan 617
    tanh 618
  memory allocation
    calloc 75
    free 238
    malloc 376
    realloc 452
  memory operations
    memchr 396
    memcmp 397
    memcpy 399
    memmove 402
    memset 403
  miscellaneous
    assert 51
    getenv 279
    longjmp 360
    perror 427
    rand 449
    setjmp 497
    srand 528
  multibyte
    wcscat 708
    wcschr 709

SAA functions *(continued)*
multibyte *(continued)*
wcscmp 711
wcscpy 715
wcscspn 717
wcslen 722
wcsncat 723
wcsncmp 725
wcsncpy 727
wcspbrk 729
wcsrchr 731
wcsspn 735
wcswcs 747
program
abort 39
atexit 54
exit 179
searching
bsearch 71
stream input/output
fclose 187
feof 197
ferror 198
fflush 199
fgetc 200
fgetpos 203
fgets 205
fprintf 224
fputc 227
fputs 230
fread 236
freopen 242
fscanf 245
fseek 247
fsetpos 249
ftell 257
fwrite 263
getc 270
getchar 270
gets 281
printf 429
putc 436
putchar 436
puts 441
scanf 486
setvbuf 508
sprintf 525
sscanf 530
tmpfile 626
ungetc 681

SAA functions *(continued)*
string
strerror 550
string manipulation
strcat 536
strchr 537
strcmp 538
strcpy 544
strcspn 546
strlen 564
strncat 566
strncmp 568
strncpy 570
strpbrk 575
strrchr 581
strspn 585
strstr 587
strtod 590
strtok 592
strtol 594
strtoll 598
strtoul 600
strtoull 602
time
asctime 47
clock 90
ctime 114
difftime 140
gmtime 289
localtime 356
mktime 410
trigonometric
acos 43
asin 49
atan 53
atan2 53
cos 96
cosh 97
sin 514
sinh 515
tan 617
tanh 618
variable argument handling
va_arg 696
va_end 696
va_start 696
vfprintf 698
vprintf 700
vsprintf 702

scanf function 486
scanf functions
  infinity and NaN values in 28
search.h include file 772
_searchenv function 492
searching
  bsearch function 71
  buffers 396
  character keys, for 342
  for file 492
  qsort 446
  strings 537, 575, 585
  strings for tokens 592
searching and sorting functions 6
seed 528
setbuf function 493
_set_crt_msg_handle function 495
setjmp function 497
setjmp.h include file 773
setlocale function 499
_setmode function 505
setting
  characters
    buffers 403
  characters in strings 574
  file modification time 693
  file translation mode 505
  floating-point control word 94
settings in child process 520
setvbuf function 508
share.h include file 773
signal function 510
signal.h include file 773
signals
  in child process 520
  list of 510
  resetting 511
  signal function
  user-defined 510
sin function 514
sine 514
sinh function 515
size of allocated memory 413
size_t 773
_sopen function 516
sorting 446
_spawn functions
  _spawnl 519
  _spawnle 519
  _spawnlp 519
_spawn functions *(continued)*
  _spawnlpe 519
  _spawnv 519
  _spawnve 519
  _spawnvp 519
  _spawnvpe 519
spawning processes 519
_splitpath function 523
sprintf function 525
square root function 527
srand function 528
_srotl function 529
_srotr function 529
sscanf function 530
stack
stack environment
  restoring 360
  saving 497
standard streams
  redirecting
    using system function 613
standard types
  FILE 775
_stat function 532
status information, files 532
status word, floating-point
  clearing 88
  getting 534
_status87 function 534
stdarg.h include file 773
stddef.h include file 773
stdio.h include file 774
stdlib.h include file 775
stopping a program 39
storage
  allocating
    temporarily 45
    with debug memory functions 124, 130
  checking blocks of 291
  freeing
  getting information about 154, 157, 656, 659
  reallocating 132
  thread-specific 623
  with debug memory functions 126
storage allocation 75
strcat function 536
strchr function 537
strcmp function 538
  comparing
    ignoring case 538

strcmpi function 540
strcoll function 542
strcpy function 544
strcspn function 546
_strdate function 548
strdup function 549
stream I/O functions 11
stream input/output (I/O)
  fclose 187
  feof 197
  ferror 198
  fflush 199
  fgetc 200
  fgets 205
  fopen 218
  fprintf 224
  fputc 227
  fputs 230
  fread 236
  freopen 242
  fscanf 245
  fseek 247
  ftell 257
  fwrite 263
  getc 270
  getchar 270
  gets 281
  printf 429
  putc 436
  putchar 436
  puts 441
  rewind 465
  scanf 486
  setbuf 493
  setvbuf 508
  sprintf 525
  sscanf 530
  tmpfile 626
  ungetc 681
  va_arg 696
  va_end 696
  va_start 696
  vfprintf 698
  vprintf 700
  vsprintf 702
streams
  association with file 194
  closing all 188
  standard
    See standard streams
_strerror function 550, 551
strfmon function 553
strftime function 557
stricmp function 562
string conversion functions
  _atold 64
  _ecvt 166
  _fcvt 192
  _gcvt 268
  infinity and NaN values in 31
  strtold 596
string functions 19
string.h include file 777
strings
  comparing 540, 546, 562, 568, 572
  concatenating 536, 566
  converting
    from floating point 268
    from integer 336
    from long integer 367
    monetary value to string 553
    multibyte to wide 389, 391
    to floating-point 56, 60, 62
    to formatted time 557
    to integer 58
    to long double 64, 596
    to long integer 60, 62
    to lowercase 565
    to uppercase 604
    wide character to double value 737
    wide character to multibyte 733, 743
    wide character to unsigned long value 745
  copying 544
  duplicating 549
  error message 551
  ignoring case 546
  initializing 570, 574
  length of 564
  printing to screen 99
  representing date as 548
  reversing 583
  searching 537, 575, 585
  searching for tokens 592
  setting characters in 574
  strerror 550
  strstr 587
  writing 230
strlen function 564

strlwr function 565
strncat function 566
strncmp function 568
strncpy function 570
strnicmp function 572
strnset function 574
strpbrk function 575
strptime function 576
strrchr function 581
strrev function 583
strset function 574
strspn function 585
strstr function 587
_strtime function 589
strtod function 590
strtok function 592
strtol function 594
strtold function 596
strtoll function 598
strtoul function 600
strtoull function 602
strxfrm function 605
supported functions 784—807
suspension of parent process 519
_swab function 608
swapping bytes 608
swprintf function 609
swscanf function 611
sys\stat.h include file 778
sys\timeb.h include file 778
sys\types.h include file 778
sys\utime.h include file 778

T

table of supported functions 784—807
tan function 617
tangent 617
tanh function 618
_tcalloc function
_tell function 619
_tempnam function 621
temporary files
  deleting with _fcloseall 188
  names 775
  number of 775
  removing 482
temporary storage, allocating 45
terminating
  a process 180
terminating *(continued)*
  a program 39, 179
  constructors and destructors 107
testing 319
  for ASCII alphabetic or underscore 325
  for ASCII alphabetic, digit, or underscore 325
  for ASCII integer value 319
text files 218
_tfree function
threads
  creating 66
  ending 169
  passing values to 66
  storage 623
_threadstore function 623
__TILED__ macro
tiled memory
time
  converting from long integer to string 114
time function 625
time zone
  setting with tzset 634
time.h include file 779
tm structure 289
_tmalloc function
TMP environment variable
  tempnam function 621
TMP_MAX 627
tmpfile function 626
  tmpfile 626
tmpnam function 627
  file names 775
tmpnam function, file names 775
_toascii function 628
tokens 592
_tolower function 628, 631
_toupper function 628, 631
towlower function 632
towupper function 632
translation mode, default in child process 520
_trealloc function
trigonometric functions 8
TZ environment variable 634
_tzset function 634

U

_uaddmem function 637

_ucalloc function 641
_ucreate function 646
_udestroy function 653
_udump_allocated function 656
_udump_allocated_delta function 659
_uheapmin function 667
_heap_walk function 298, 671
_ulltoa function
_ultoa function 675, 676
umalloc.h include file 779
_umask function 679
underflow error 378
ungetc function 681
_ungetch function 683
ungetwc function 685
_unlink function 688
_uopen function 689
updating files 194
_ustats function 691
_utime function 693

V

va_arg function 696
va_end function 696
va_start function 696
variable argument functions 16
variables
  environment
    modifying from within program 439
    TMP 621
    using in search 492
  global
    P_tmpdir 621
vfprintf function 698
VisualAge for C++
  extensions to SAA
  predefined macros 808
  publications 823
  supported functions 784—807
vprintf function 700
vsprintf function 702
vswprintf function 704

W

_wait function
wchar.h include file 780
wcrtomb function 706
wcscat function 708
wcschr function 709
wcscmp function 711
wcscoll function 713
wcscpy function 715
wcscspn function 717
wcsftime function 719
wcsid function 721
wcslen function 722
wcsncat function 723
wcsncmp function 725
wcsncpy function 727
wcspbrk function 729
wcsrchr function 731
wcsrtombs function 733
wcsspn function 735
wcsstr function 736
wcstod function 737
wcstok function 739
wcstol function 741
wcstombs function 743
wcstoul function 745
wcstr.h include file 782
wcswcs function 747
wcswidth function 748
wcsxfrm function 749
wctob function 752
wctomb function 753
wctype function 755
wctype.h include file 782
wcwidth function 758
WEOF macro 781
wide character conversions
  date and time 719
  to double value 737
  to integer value 741
  to multibyte character 706, 753
  to multibyte string 733, 743
  to single-byte character 752
  to unsigned long value 745
  to upper case 632
wide character string functions 21
write from buffer, file 759
_write function 759
writing
  error messages 551

Y

_y0,_y1,_yn (bessel functions) 69

Communicating Your Comments to IBM

IBM VisualAge for C++ for Windows
C Library Reference

Version 3.5

Publication No. S33H-5038-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

If you prefer to send comments by mail, use the RCF at the back of this book.

If you prefer to send comments by FAX, use this number:

- United States and Canada: 416-448-6161
- Other countries: (+1)-416-448-6161

If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.

- Internet: torrcf@vnet.ibm.com
- IBMLink: toribm(torrcf)
- IBM/PROFS: torolab4(torrcf)
- IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

IBM VisualAge for C++ for Windows
C Library Reference

Version 3.5

Publication No. S33H-5038-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
S33H-5038-00

Cut or Fold
Along Line

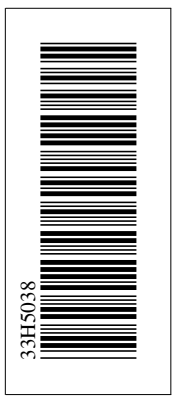
Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Part Number: 33H5038
Program Number: 33H4979
33H4980

Printed in U.S.A.