IBM VisualAge C++ for OS/2

S25H-6958-00

Programming Guide

Version 3.0

IBM VisualAge C++ for OS/2

S25H-6958-00

Programming Guide

Version 3.0

Note!

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

Third Edition (May 1995)

This edition applies to Version 3.0 of IBM VisualAge C++ for OS/2 (Programs 30H1664, 30H1665, and 30H1666) 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. Consult the latest edition of the applicable IBM system bibliography for current information on this 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. Please 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.

Contents

Notices

Programming Interface Information

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

General-Use programming interfaces allow the customer to write programs that obtain the services of the VisualAge C++ compiler, debugger, browser, execution trace analyzer, visual builder, editor, data access frameworks, and class libraries.

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.

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:

BookManager	Personal System/2
C/2	PS/2
C Set/2	Presentation Manager
C Set ++	Systems Application Architecture
Common User Access	SAA
CUA	VisualAge
IBM	WorkFrame
LibraryReader	Workplace Shell
Open Class	System Object Model
Operating System/2	SOM
OS/2	
OS/2 Warp	

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.

About This Book

This book describes coding techniques such as multithreading, creating DLLs, using templates, signal and exception handling, managing memory, and creating programs that use 16-bit and 32-bit code. The book focuses mostly on the C and C++ techniques involved, rather than the lower-level OS/2 techniques that are described in the *Control Program Guide and Reference*.

Use this book with the other publications described in the “Bibliography” on page 429.

Who Should Read This Book

This book is written for application and systems programmers who want to use IBM VisualAge C++ for OS/2 to develop and run C or C++ applications. You should have a working knowledge of the C or C++ programming language, the OS/2 operating system, and other products described in *Welcome to VisualAge C++*.

How to Use This Book

For an overview and tour of VisualAge C++, see the *Welcome to VisualAge C++*. For introductory information on how to use the VisualAge C++ compiler and tools to compile, link, debug, browse, and trace your program, see the *User's Guide*. For reference information on the more technical aspects of the compiler and advanced programming techniques, use this book.

How to Read the Syntax Diagrams

This book uses two methods to show syntax. One is for commands, preprocessor directives, and statements; the other is for compiler options.

Syntax for Commands, Preprocessor Directives, and Statements

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►—— symbol indicates the beginning of a command, directive, or statement.

  The ——► symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ►—— symbol indicates that a command, directive, or statement is continued from the previous line.

  The ——►◄ symbol indicates the end of a command, directive, or statement.

How to Read Syntax Diagrams

Diagrams of syntactical units other than complete commands, directives, or statements start with the ►── symbol and end with the ──► symbol.

Note: In the following diagrams, `STATEMENT` represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

```
►►──STATEMENT──required_item──────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──STATEMENT──┬──────────────┬───────────────────────────────►◄
               └─optional_item─┘
```

- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──STATEMENT──┬─required_choice1─┬───────────────────────────►◄
               └─required_choice2─┘
```

  If the items are optional, the entire stack appears below the main path.

```
►►──STATEMENT──┬──────────────────┬───────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```

  The item that is the default appears above the main path.

```
               ┌─default_item───┐
►►──STATEMENT──┴─alternate_item─┴─────────────────────────────►◄
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
               ┌─────────────────┐
►►──STATEMENT──▼─repeatable_item─┴────────────────────────────►◄
```

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, **`pragma`**).

  Variables appear in italicized lowercase letters (for example, *`identifier`*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Note: The white space is not always required between tokens, but it is recommended that you include at least one blank between tokens unless specified otherwise.

The following syntax diagram example shows the syntax for the **`#pragma comment`** directive. (See the *Language Reference* for information on the **`#pragma`** directive.)

```
 1  2   3      4        5        6                                  9    10
►►─#──pragma──comment──(─┬──────compiler─────────────────────────┬─)─►◄
                         ├──────date─────────────────────────────┤
                         ├──────timestamp────────────────────────┤
                         └──┬───copyright──┬──┬──────────────────┤
                            └───user───────┘  └──,─"characters"──┘
                                                  7      8
```

The syntax diagram is interpreted in the following manner:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword **`pragma`** must appear following the # symbol.

4 The keyword `comment` must appear following the keyword **`pragma`**.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler, date, timestamp, copyright,` or `user`.

7 If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Syntax for Compiler Options

- Optional elements are enclosed in square brackets [ ].
- When you have a list of items from which you can choose one, the logical OR symbol (|) separates the items.
- Variables appear in italicized lowercase letters (for example, *num*).

Examples

Syntax	Possible Choices
/L[+|-]	/L /L+ /L-
/Lt"*string*"	/Lt"Listing File for Program Test"

Note that, for options that use a plus (+) or minus (-) sign, if you do not specify a sign, the plus is assumed. For example, the /L and /L+ options are equivalent.

Icons Used in This Book

The VisualAge 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.

This icon identifies information that applies *only* to the C++ language.

Changes to this Book

In addition to rewriting and re-organizing some of the existing information in the book to improve clarity and ease of use, we've added some brand new chapters. Some describe new components, others describe functional enhancements we've made to previously existing components.

Important: For details on late changes, features, and restrictions, please ensure you read the README file.

New Product Features You'll Find in this Book

New features that involve support from many components include:

- **Direct-to-SOM (DTS) support**

  The IBM System Object Model (SOM) provides a common programming interface for building and using objects. SOM improves your C++ programming productivity in two ways:

  – You can release new versions of a class library without requiring users of the library to recompile their applications.
  – You can make your C++ classes and objects accessible to programs written in other languages, and write C++ programs that use classes and objects created using other SOM-supported languages.

  You can make classes and methods in existing C++ programs SOM-accessible without having to rewrite class and method definitions. Although SOM imposes some restrictions on C++ coding conventions, you should be able to convert most C++ programs for SOM support with minimal effort. Compiler options, **#pragma** directives, and other elements have been added to support DTS compilation.

  For more details on DTS support, see Chapter 16, "The IBM System Object Model" on page 277

- **Improved memory management, including user heaps**

  In addition to improving our memory allocation methods to increase speed and reduce memory waste, we have added functions that you can use to create and manage your own heaps of memory. You can use these user heaps in addition to, or in place of, the regular runtime heap.

  Debug versions of memory management functions are now available for user heaps and for tiled memory, as well as for regular memory. There are also new heap-checking functions, similar to those provided by other C/C++ compilers, that you can use to debug your memory management problems.

  The debugger has also added heap-checking capabilities. When your application stops, you can check the dynamic variables allocated by memory management functions on the heap.

  Memory management is discussed in Chapter 15, "Managing Memory" on page 253, and the new functions are described in the *C Library Reference*. For more details on the debugger, see the section on debugging in the *User's Guide*, or the online help for the debugger.

Changes

- **Support for POSIX locales and functions**

  A *locale* is a collection of data that encodes information about the cultural environment. Locales provide a way to internationalize your applications by defining the language, character sets, date and time format, and other culturally-determined elements.

  VisualAge C++ has added support for locales based on the IEEE POSIX P1003.2 and X/Open Portability Guide standards for global locales and coded character set conversion. To support locales, we have added a number of new functions, header files, environment variables, and utilities. The functions include a number of multibyte functions proposed for addition to the ANSI/ISO C standard.

  For more information about locales and how to use them, see the Chapter 7, “Introduction to Locale” on page 91. The new functions are described in the *C Library Reference*.

- **Template Resolution Improvements**

  We've added a new 32-bit linker, VisualAge C++ linker, to complement our 32-bit compiler. VisualAge C++ linker is faster than our previous linker, LINK386, and provides additional features such as:

  - Template resolution independent of the compiler (meaning you no longer have to invoke the linker through `icc` when you use templates).
  - Linker optimization options for improving resolution of function and member function calls. These options can significantly reduce the size of your application files.
  - Packing of debug information to reduce the size of your files and potentially improve debugger performance.

  The VisualAge C++ linker syntax is consistent with that of LINK386 to make the transition easier. It also supports most LINK386 options along with the new VisualAge C++ linker options.

  For more information about templates, see Chapter 9, “Using Templates in C++ Programs” on page 133. For more information about VisualAge C++ linker, see the section on Linking in the *User's Guide*.

- **Optimization Improvements**

  VisualAge C++ has a reputation for producing very fast executables. We've increased and enhanced the optimizations in our IBM VisualAge C++ Version 3.0 for OS/2 compiler to further improve your program's performance.

  We also realize that in some cases, the size of your executable files may be more critical than the speed they run at. For this reason, both the compiler and VisualAge C++ linker linker have implemented options that reduce the size of your code, while still providing some measure of performance improvement. For more information about optimization and the options that control it, see Chapter 4, “Optimizing Your Program” on page 35 in this book and the sections on compiling and linking in the *User's Guide*.

Summary of Changes to this Book

Here is a summary of the changes discussed above which have been made to this Book:

Added

Several chapters on new topics have been added:

- Customizing the language and culture sensitive behavior of applications, in Chapter 7, “Introduction to Locale” on page 91 and in Chapter 8, “Building a Locale” on page 113
- Managing the memory of your application more efficiently, see Chapter 15, “Managing Memory” on page 253
- Creating and using language-independent objects, see Chapter 16, “The IBM System Object Model” on page 277

Moved

One chapter and one appendix have been moved to other documents in the IBM VisualAge C++ for OS/2 library:

- The chapter on compiling and linking your program, including using debugging and diagnostic options, has been moved to the *User's Guide*.
- The appendix that dealt with solving common C problems has been moved to *Frequently-Asked Questions*.

Rewritten

Two chapters have been extensively re-written to improve clarity and reading ease.

- Using compiler-generated templates in your programs, see Chapter 9, “Using Templates in C++ Programs” on page 133.
- How to use 16-bit code with the 32-bit code generated by VisualAge C++, see Chapter 12, “Calling between 32-Bit and 16-Bit Code” on page 191.

C and C++ Language Standards and Portability

The VisualAge C++ product is designed according to the specifications of the *American National Standard for Information Systems / International Standards Organization – Programming Language C*, ANSI/ISO 9899-1990[1992], as understood and interpreted by IBM as of October 1993. Behavior that the ANSI C Standard declares as implementation-defined is described in Appendix A, "ANSI Notes on Implementation-Defined Behavior" on page 339.
If you will be using VisualAge C++ to develop code according to the American National Standards Institute (ANSI) standard, you should also refer to the ANSI guidelines. If you will be developing code according to the International Standards Organization (ISO) standard, refer to the ISO guidelines. General information about writing portable C code is included in the *Portability Guide for IBM C*, SC09-1405.

VisualAge C++ also implements the Systems Application Architecture (SAA) C Level 2 definition, which is a superset of the ANSI standard. For more information on the SAA C standard, see the *Language Reference*. If you will be using VisualAge C++ to develop C applications to be compiled and run on other Systems Application Architecture* (SAA*) systems, you should follow the SAA standards as outlined in the *SAA Common Programming Interface C Language Reference – Level 2*, SC09-1308.

When following ANSI, ISO, or SAA standards, do **not** use the extensions specific to VisualAge C++ compiler as described in the *C Library Reference* and the *Language Reference*.

At this time, there is no universal standard for the C++ language comparable to C standards. However, an ANSI committee is developing a C++ language standard. Its September 17, 1992 working paper, *Draft Proposed American National Standard for Information Systems — Programming Language C++*, X3J16/92-0060, was used as a base document for developing the VisualAge C++ compiler. The VisualAge C++ compiler will continue to change its design in accordance with the ANSI standard as it evolves. If portability of your C++ programs is important, isolate those parts of your code that use the Collection and User Interface class libraries, which are specific to VisualAge C++ product. Then you can easily remove or replace them when migrating your programs.

If you will be using IBM VisualAge C++ for OS/2 for the development of applications that will run only under the OS/2 operating system, you may want to exploit the OS/2 services and APIs and the VisualAge C++ multithread features. For details of multithread features available under VisualAge C++, see Chapter 5, "Creating Multithread Programs" on page 47).

How to Get Help

There are three kinds of online information available to you while you are using VisualAge 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 C++. For your convenience, the online documents are presented in two formats:

- Standard format. See "Getting Help Inside VisualAge C++" on page xxiv for instructions on opening standard format documents from inside VisualAge C++. See "Getting Help from the Command Line" on page xxv for instructions on opening standard format documents from the command line. The following documents are available in standard format:
  - *Read Me First!*
  - *Welcome to VisualAge C++*
  - *User's Guide*
  - *Programming Guide*
  - *Visual Builder User's Guide*
  - *Visual Builder Parts Reference*
  - *Building VisualAge C++ Parts for Fun and Profit*
  - *Open Class Library User's Guide*
  - *Open Class Library Reference*
  - *Language Reference*
  - *C Library Reference*
  - *Editor Command Reference*
- BookManager format. See "BookManager Books" on page xxv for details on how to access online documents in this format. For a list of the VisualAge C++ documents that are available in BookManager format, see "Bibliography" on page 429.

Contextual help
Contextual help is available throughout VisualAge 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 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 C++, as well as individual task lists for each of its components.

Getting Help Inside VisualAge C++

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

- To get general contextual help for the component of VisualAge 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 C++ being used
- To get detailed information, open the **Information** folder in the VisualAge C++ folder. In this folder you will find icons for a variety of online documents that describe, in detail, the different aspects of VisualAge C++. To open a particular online document, double click on its icon.

Getting Help from the Command Line

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

```
VIEW 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:

```
VIEW CPPLNG.INF OPERATOR PRECEDENCE
```

BookManager Books

In addition to standard format, the online documents are also available in BookManager format. In this format they can be read using the BookManager READ/2 product (program number 73F6023). VisualAge C++ comes complete with the IBM Library Reader, which allows you to read BookManager books without having to install the complete BookManager READ/2 product. Like the standard format, the BookManager format features hypertext links and a search utility.

Part 1. Running Your Program

This part describes how to set environment variables for running your program, how to specify runtime options, and how to redirect standard input/output.

Running Your Program

1 Setting Runtime Environment Variables

This chapter discusses environment variables which compose the runtime of the applications you build. You need to be aware of them when you build the application as they can affect the behaviour of your application and can lead to unexpected results if not taken into account.

How to Set Environment Variables

You can set the runtime environment for your application by using OS/2 environment variables. You can set most of them from the command line, in your CONFIG.SYS file, in a command file using the `SET` command, or from within your program using the `putenv` function.

You can put an optional semicolon at the end of the commands that set the environment variables so that you can later append values to the variables from the command line.

The functions that access these environment variables are not available when you use the subsystem libraries. To access the environment variables when you are using the subsystem libraries, you must use OS/2 APIs. See the online *PM Programming Reference* for more information about OS/2 APIs.

Some of the variables discussed in this chapter are also used at compile time. The compiler environment variables are described in the *User's Guide*. For more information on environment variables in general, see the OS/2 *Master Help Index* or *OS/2 Command Reference*.

Application Environment Variables

The following environment variables determine where your application will look to locate files necessary to the execution of its tasks. Files for such things as command interpretation, runtime messages, and locale settings will all be needed. If you do not make sure that the environment variables are correctly set, your application may fail to find a file, or worse yet, find and use an unintended file.

Application Environment Variables

The following environment variables are discussed:

- COMSPEC
- DPATH
- LANG
- LC Environment Variables
- LIBPATH
- LOCPATH
- PATH
- TEMPMEM
- TMP
- TZ

COMSPEC The `system` function uses this variable to locate the command interpreter. When the OS/2 operating system is installed, the installation program sets the COMSPEC variable in the CONFIG.SYS file to the name and path of the command interpreter. To change the COMSPEC variable, use the `SET` command in CONFIG.SYS. For example:

```
SET COMSPEC=c:\mydir\mycmd.exe
```

sets the command interpreter as `mycmd.exe` in the `c:\mydir` directory. 🕮 For more information on the `system` function, refer to the *C Library Reference*.

DPATH The `iconv` function uses this environment variable to locate the tables built by the `genxlt` utility. Also, when you run an executable program, unless you have bound the runtime messages files to the executable file using the `MSGBIND` utility, the runtime messages files must be either in your current directory or in one of the directories specified by the DPATH variable.

The DPATH variable can be set by using the `SET` command.

For example, given the following DPATH value:

```
DPATH=c:\kevin;d:\michel
```

the program would search the current directory, and then the directories `c:\kevin` and `d:\michel`, in that order.

If when you installed VisualAge C++ you chose the option of having your CONFIG.SYS file modified, the DPATH statement in your CONFIG.SYS file was changed to:

```
SET DPATH=d:\IBMCPP\HELP;d:\IBMCPP;d:\IBMCPP\LOCALE;d:\IBMCPP\MACROS
```

Application Environment Variables

LANG

The `LANG` environment variable specifies the default locale name for the locale categories, when the `LC_ALL` environment variable is not defined and the locale categories environment variable is not defined.

LC Environment Variables

The following environment variables are used to specify the names of locale categories:

- `LC_ALL`
- `LC_COLLATE`
- `LC_CTYPE`
- `LC_MESSAGES`
- `LC_MONETARY`
- `LC_NUMERIC`
- `LC_TIME`
- `LC_TOD`
- `LC_SYNTAX`

"Locale Source Files" on page 120 describes the locale categories that correspond to these environment variables. "Customizing a Locale" on page 103 tells you how to use these environment variables to customize a locale.

LIBPATH

The operating system searches the directories specified by this directive to find all `.DLL` files required by the program. The LIBPATH is set at system startup and cannot be reset dynamically. The library DLLs and any user DLLs must be in one of the directories specified by the LIBPATH.

This variable can only be specified in the CONFIG.SYS file. For example:

```
LIBPATH=.;c:\cmlib;c:\IBMCPP\DLL;
```

sets the DLL search path to the current directory, `c:\cmlib`, and `c:\IBMCPP\DLL`. For more information on DLLs, see Chapter 6, "Building Dynamic Link Libraries" on page 61.

LIBPATH **cannot** be specified using the `SET` command. Also, unlike the PATH environment variable, the operating system does not check the current directory first by default. If you want the current directory checked first, you must explicitly list it in the LIBPATH statement.

LIBPATH can be useful in switching between conflicting versions of a tool. Simply create an empty DLL directory somewhere in the LIBPATH and copy the required DLLs into this directory to control version execution.

Note: While the LIBPATH variable itself cannot be reset without rebooting the system, OS/2 Warp Version 3.0 provides a mechanism for you or an application to *extend* the path variable. Applications can use `DosSetExtLIBPATH` to set the path extension and `DosQueryExtLIBPATH` to query the current extension. Parameters of these functions tell the system whether the extension should go before or after the LIBPATH.

For more information on `DosSetExtLIBPATH` and `DosQueryExtLIBPATH`, see the OS/2 Warp Version 3.0 *Control Program Guide and Reference*.

LOCPATH The `setlocale` function uses this environment variable at run time to search for locale information not in the current directory.

If you chose to modify your CONFIG.SYS file at install time, your LOCPATH statement in that file will look like:

```
    SET LOCPATH=D:\IBMCPP\LOCALE
```

PATH The `system`, `exec`, and `_spawn` functions use this environment variable to search for `.EXE` and `.CMD` files not in the current directory. You can set it by using the `SET` command from an OS/2 window or fullscreen session, or in a command file. For example,

```
   SET PATH=c:\IBMCPP\BIN;c:\IBMCPP\HELP, e:\ian;d:\steve
```

You can specify one or more directories with this variable. Given the above example, the path searched would be the current directory and then the directories `c:\IBMCPP\BIN`, `c:\IBMCPP\HELP`, `e:\ian`, and `d:\steve`.

For further information on the functions that use PATH, refer to the *C Library Reference*.

TEMPMEM Use this variable to control whether temporary files are created as memory files or as disk files. It can be set using the `SET` command either in the CONFIG.SYS file or on the command line. For example:

```
   SET TEMPMEM=on
```

If the value specified is `on` (in upper-, lower-, or mixed case), and you compile with the `/Sv+` option, the temporary files will be created as memory files. If TEMPMEM is set to any other value, the temporary files will be disk files. If you do not compile with `/Sv+`, memory file support is not available and your program will end with an error when it tries to open a memory file.

If TEMPMEM will be used by a program, you must set its value in the environment before the program starts. You cannot set it from within the program.

TMP

The directory specified by this variable holds temporary files, such as those created using the `tmpfile` If any of these values is not valid, function. (`tmpfile` is described in the *C Library Reference*.) You must set the TMP variable to use the VisualAge C++ compiler.

Set the TMP variable with the `SET` command either in the CONFIG.SYS file or on the command line. For example:

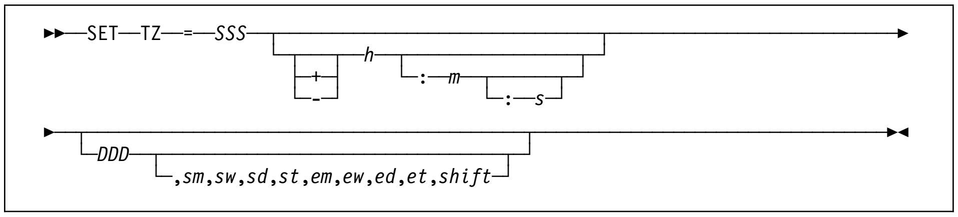
```
SET TMP=c:\IBMCPP\TMP
```

You can specify only one directory using the TMP variable.

Note: The TMP environment variable may be set by applications other than VisualAge C++. If another application changes TMP to specify a different directory, this directory will be used by VisualAge C++ to hold temporary files.

TZ

This variable is used to describe the timezone information that the locale will use. To set TZ, use the `SET` which has the following format:



```
►►──SET──TZ──=──SSS──┬─────────────────────────────┬────────────────────►
                     └─┬───┬─h─┬─────────────────┬─┘
                       ├─+─┤   └─:─m─┬───────┬───┘
                       └─-─┘         └─:─s───┘

►──┬────────────────────────────────────────────┬─────────────────────►◄
   └─DDD─┬────────────────────────────────────┬─┘
         └─,sm,sw,sd,st,em,ew,ed,et,shift─────┘
```

The values for the TZ variable are defined below. The default values given are for the built-in "C" locale defined by the ANSI C standard.

Figure 1 (Page 1 of 2). TZ Environment Variable Parameters

Variable	Description	Default Value
SSS	Standard-timezone identifier. It must be three characters, must begin with a letter, and can contain spaces.	EST
h, *m*, *s*	The variable *h* specifies the difference (in hours) between the standard time zone and coordinated universal time (CUT), formerly Greenwich mean time (GMT). You can optionally use *m* to specify minutes after the hour, and *s* to specify seconds after the minute. A positive number denotes time zones west of the Greenwich meridian; a negative number denotes time zones east of the Greenwich meridian. The number must be an integer value.	5

Figure 1 (Page 2 of 2). TZ Environment Variable Parameters

Variable	Description	Default Value
`DDD`	Daylight saving time (DST) zone identifier. It must be three characters, must begin with a letter, and can contain spaces.	EDT
`sm`	Starting month (1 to 12) of DST.	4
`sw`	Starting week (-4 to 4) of DST. Use negative numbers to count back from the last week of the month (-1) and positive numbers to count from the first week (1).	1
`sd`	Starting day of DST. 0 to 6 if `sw` != 0 1 to 31 if `sw` = 0	0
`st`	Starting time (in seconds) of DST.	3600
`em`	Ending month (1 to 12) of DST.	10
`ew`	Ending week (-4 to 4) of DST. Use negative numbers to count back from the last week of the month (-1) and positive numbers to count from the first week (1).	-1
`ed`	Ending day of DST. 0 to 6 if `ew` != 0 1 to 31 if `ew` = 0	0
`et`	Ending time of DST (in seconds).	7200
`shift`	Amount of time change (in seconds).	3600

For example:

```
SET TZ=CST6CDT
```

sets the standard time zone to CST, the daylight saving time zone to CDT, and sets a difference of 6 hours between CST and CUT. It does not set any values for the start and end date of daylight saving time or the time shifted.

When TZ is not present, the default is `EST5EDT`, the "C" locale value. When only the standard time zone is specified, the default value of *n* (difference in hours from GMT) is `0` instead of 5.

If you give values for any of *sm, sw, sd, st, em, ew, ed, et*, or *shift*, you must give values for all of them. the entire statement is considered not valid, and the time zone information is not changed.

The value of TZ can be accessed and changed by the `tzset` function. For more information on `tzset`, see the *C Library Reference*.

Application Environment Variables

Running Your Program

This chapter describes common tasks associated with the running of your application. These include: declaring arguments to main, passing data to your program, returning values from main, expanding global filenames, and redirecting standard streams.

Choice of User Interfaces for Running an Application

You can compile and link your files to create an executable unit through any one of three user interfaces:

- using the WorkFrame environment to create a project,
- using the WorkPlace Shell interface to create an object, or simply,
- using the command line to create an executable file.

Directions on how to create and run these elements through the various interfaces is discussed in the *User's Guide*.

Note: The OS/2 operating system uses the PATH environment variable to find executable files. You can run a program from any directory, as long as the executable program is either:

- In your current working directory
- In one of the directories specified by the PATH environment variable
- Specified on the command line with a fully qualified path name

If more than one of these directories contains an executable file with the same name, the first executable file that is found on the path is run.

The run time messages files (*CPPO.MSG* for the C run time) must also be either in your current working directory or in one of the directories specified by the DPATH environment variable, unless you have bound the messages to your executable file using the MSGBIND utility. The utility is described in the *User's Guide*.

You can the `system` function in the VisualAge C++ runtime library to run other programs and OS/2 commands from within a program. See the *C Library Reference* for more information on the `system` function.

Declaring Arguments to main.

To set up your program to receive data from the command line, the project parameter page, or through object conversations, declare arguments to **main** as:

```
int main(int argc, char **argv, char **envp)
```

Note: In the ANSI definition of C, there are only two possible ways to define **main**:

```
int main(int argc, char **argv) { ... }
```

and

```
int main(void) { ... }
```

VisualAge C++ supports this definition as well as the ANSI-conforming definitions.

By declaring these variables as arguments to **main**, you make them available as local variables. You need not declare all three arguments, but if you do, they must be in the order shown. To use the `envp` argument, you must declare `argc` and `argv`, even if you do not use them.

Each OS/2 command-line argument, regardless of its data type, is stored as a null-terminated string in an array of strings. The command is passed to the program as the `argv` array of strings. The number of arguments appearing at the command prompt is passed as the integer variable `argc`.

The first argument of any command is the name of the program to run. The program name is the first string stored at `argv[0]`. Because you must always give a program name, the value of `argc` is at least 1.

The runtime initialization code stores the first argument after the program name at `argv[1]`, the second at `argv[2]`, and so on through the end of the arguments. The total number of arguments, including the program name, is stored in `argc`. The `argv[argc]` is set to a `NULL` pointer.

You can also access the values of the individual arguments from within the program using `argv`. For example, to access the value of the last argument, use the expression `argv[argc-1]`.

The third argument passed to **main**, `envp`, is a pointer to the environment table. You can use this pointer to access the value of the environment settings. (Note that the `getenv` function accomplishes the same task and is easier to use.) The `envp` argument is not available when you use the subsystem libraries.

The `putenv` routine may change the location of the environment table in storage, depending on storage requirements; because of this, the value given to `envp` when you start to run your program might not be correct throughout the running of the program. The `putenv` and `getenv` functions access the environment table correctly, even when its location changes. 🕮 For more information about `putenv` and `getenv`, see the *C Library Reference*.

Passing Data to a Program

To pass data to your program by way of the command line, give one or more arguments after the program name. Each argument must be separated from other arguments by one or more spaces or tab characters. You must enclose in double quotation marks any arguments that include spaces, tab characters, double quotation marks, or redirection characters. For example:

```
hello 42 "de f" 16
```

This command runs the program named `hello.exe` and passes three arguments: `42`, `de f`, and `16`. The combined length of all arguments in the command (including the program name) cannot exceed the OS/2 maximum length for a command. For information on the maximum allowable command length, see the *OS/2 Commmand Reference*.

You can also use escape sequences within arguments. For example, to represent double quotation marks, precede the double quotation character with a backslash. To represent a backslash, use two backslashes in a row. For example, when you invoke the `hello.exe` program from the preceding example with this command:

```
hello "ABC\"" \"HELLO\\
```

the arguments passed to the program are `ABC"` and `"HELLO\`.

Returning Values from main

The function **main**, like any other C or C++ function, returns a value. Its return value is an `int` value that is passed to the operating system as the return code of the program that has been run.

You can check this return code with the IF ERRORLEVEL command in OS/2 batch files. 🕮 For more information on the IF ERRORLEVEL command, see the OS/2 online *Command Reference*

To cause **main** to return a specific value to the operating system, use the `return` statement or the `exit` function to specify the value to be returned. The statement

```
return 6;
```

returns the value 6. For instance, in a REXX program the return value is returned in the RC variable.

```
/* RET6.C */
int main(void) {return 6;}

/* Checkret.CMD */
'RET6'
say 'RC='rc
```

will output RC=6 when run.

If you do not use either method, the return code is undefined.

For more information about **main**, see the *Language Reference*.

Expanding Global File-Name Arguments

You can expand global file-name arguments from the OS/2 command line using the OS/2 global file-name characters (or wildcard characters), the question mark (?), and asterisk (*), to specify the file-name and path-name arguments at the command prompt. To use them, you must link your program with the special routine contained in SETARGV.OBJ. module This object file is included with the libraries in the LIB directory under the main VisualAge C++ directory. If you do not link your program with SETARGV.OBJ, the compiler treats the characters literally.

SETARGV.OBJ expands the global file-name characters in the same manner that the OS/2 operating system does. (See the OS/2 *Master Help Index* for more information.) For example, when you link hello.obj with SETARGV.OBJ:

```
ILINK /NOE hello SETARGV;
```

and run the resulting executable module hello.exe with this command:

```
hello *.INC ABC? "XYZ?"
```

the SETARGV function expands the global file-name characters and causes all file names with the extension **.INC** in the current working directory to be passed as arguments to the hello program. Similarly, all file names beginning with ABC followed by any one character are passed as arguments. The file names are sorted in lexical order.

If the SETARGV function finds no matches for the global file-name arguments, for example, if no files have the extension **.INC**, the argument is passed literally.

Because the "XYZ?" argument is enclosed in quotation marks, the expansion of the global file-name character is suppressed, and the argument is passed literally as XYZ?.

Alternatively, if you use access your executables through the project interface and you frequently use global file-name expansion, you can place the SETARGV.OBJ routine in the standard libraries you use. Then the routine is automatically linked with your program.

Use the ILIB utility to delete the module named SETUPARG module from the library (the module name is the same in all VisualAge C++ libraries), and add the SETARGV module. When you replace SETUPARG with SETARGV, global file-name expansions are performed automatically on command-line arguments.

📖 For more information on the ILIB utility, see the *User's Guide*.

Redirecting Standard Streams

A C or C++ program has standard streams associated with it. You need not open them; they are automatically set up by the runtime environment when you include **<stdio.h>**. The three standard streams are:

stdin	The input device from which your program normally retrieves its data. For example, the library function getchar uses **stdin**.
stdout	The output device to which your program normally directs its output. For example, the library function printf uses **stdout**.
stderr	The output device to which your program directs its diagnostic messages.

The streams **stdprn** and **stdaux** are reserved for use by the OS/2 operating system and are not supported by VisualAge C++ compiler.

On input and output operations requiring a file pointer, you can use **stdin**, **stdout**, or **stderr** in the same manner as you would a regular file pointer.

The Presentation Manager (PM) interface uses the **stdout** and **stderr** streams somewhat differently than non-Presentation Manager programs. Strings written to **stdout** or **stderr** do not show up on the screen unless redirected. For more information on stream behaviour under Presentation Manager, see “I/O Considerations When You Use Presentation Manager” on page 33.

Redirecting Standard Streams

When a C++ program uses the I/O Stream classes, the following predefined streams are also provided in addition to the standard streams:

cin	The standard input stream.
cout	The standard output stream.
cerr	The standard error stream. Output to this stream is unit-buffered. Characters sent to this stream are flushed after each insertion operation.
clog	Also the standard error stream. Output to this stream is fully buffered.

The **cin** stream is an `istream_withassign` object, and the other three streams are `ostream_withassign` objects. These streams and the classes they belong to are described in detail in the *Open Class Library Reference*.

There may be times when you want to redirect a standard stream to a file. The following sections describe methods you can use for C and C++ programs.

Redirection from within a Program

To redirect C standard streams to a file from within your program, use the `freopen` library function. For example, to redirect your output to a file called `pia.out` instead of **stdout**, code the following statement in your program:

```
   freopen("pia.out", "w", stdout);
```

For more information on `freopen`, refer to the *C Library Reference*.

You can reassign a C++ standard stream to another `istream` (**cin** only) or `ostream` object, or to a `streambuf` object, using the `operator=`. For example, to redirect your output to a file called `michael.out`, create `michael.out` as an `ostream` object, and assign **cout** to it:

```
   #include <fstream.h>

   int main(void)
   {
      cout << "This is going to the standard output stream" << endl;

      ofstream outfile("michael.out");
      cout = outfile;
      cout << "This is going to michael.out file" << endl;

      return 0;
   }
```

You can also assign **cout** to `outfile.rdbuf()` to perform the same redirection.

📖 For more information on using C++ standard streams, see the *Open Class Library Reference*.

Redirection from the Command Line

To redirect a C or C++ standard stream to a file from the command line, use the standard OS/2 redirection symbols.

For example, to run the program `bill.exe`, which has two required parameters `XYZ` and `123`, and redirect the output from **stdout** to a file called `bill.out`, you would use the following command:

```
bill XYZ 123 > bill.out
```

You can also use the OS/2 file handles to redirect one standard stream to another. For example, to redirect **stderr** to **stdout**, you would use the command:

```
2 > &1
```

You cannot use redirection from the command line for memory files.

📖 Refer to the OS/2 online *Master Help Index* for more information on redirection symbols.

Redirecting Standard Streams

Part 2. Coding Your Program

This part describes different features of the VisualAge C++ compiler that you may want to use when you code your program, including the input and output methods, the support for multithread programs and dynamic link libraries, and ways to improve program performance and to reduce program size.

Coding Your Program

3 Performing Input/Output Operations

This chapter describes input and output methods for the VisualAge C++ compiler.

Note that record level I/O is not supported.

Using Standard Streams

Three standard streams are associated with the C language, **stdin**, **stdout**, and **stderr**. In C++, when you use the I/O Stream Library, there are four additional C++ standard streams, **cin**, **cout**, **cerr**, and **clog**. All of the standard streams are described in "Redirecting Standard Streams" on page 15.

An OS/2 file handle is associated with each of the streams as follows:

File Handle	C Stream	C++ Stream
0	**stdin**	**cin**
1	**stdout**	**cout**
2	**stderr**	**cerr**, **clog**

Note: Both **cerr** and **clog** are standard error streams; **cerr** is unit buffered and **clog** is fully buffered.

The file handle and stream are not equivalent. There may situations where a file handle is associated with a different stream, for example, where file handle 2 is associated with a stream other than **stderr**, **cerr**, or **clog**. Do not code your program in so that it is dependent on the association between the stream and the file handle.

The standard streams are not available when you are using the subsystem libraries.

For information on I/O streams when using Presentation Manager, see "I/O Considerations When You Use Presentation Manager" on page 33.

The streams **stdprn** and **stdaux** are reserved for use by the OS/2 operating system and are not supported by VisualAge C++.

Note: The C++ streams do not support the use of *data definition names* (ddnames). See the *Open Class Library Reference* for more information about the C++ streams.

Stream Processing

Input and output are mapped into logical data streams, either text or binary. Streams present a consistent view of file contents, independent of the underlying file system.

Text Streams

Text streams contain printable characters and control characters organized into lines. Each line consists of zero or more characters and ends with a new-line character (`\n`). A new-line character is not automatically appended to the end of the file.

The VisualAge C++ compiler may add, alter, or ignore some new-line characters during input or output so that they conform to the conventions for representing text in an OS/2 environment. Thus, there may not be a one-to-one correspondence between the characters in a stream and those in the external representation. See page 23 for an example of the difference in representations.

Data read from a text stream is equal to the data that was written if it consists only of printable characters and the horizontal tab, new-line, vertical tab, and form-feed control characters.

On output, each new-line character is translated to a carriage-return character, followed by a line-feed character. On input, a carriage-return character followed by a line-feed character, or a line-feed character alone is converted to a new-line character.

If the last operation on the stream is a read operation, `fflush` discards the unread portion of the buffer. If the last operation on the stream is a write operation, `fflush` writes out the contents of the buffer. In either case, `fflush` clears the buffer.

The `ftell`, `fseek`, `fgetpos`, `fsetpos`, and `rewind` functions cannot be used to get or change the file position within character devices or OS/2 pipes.

The C standard streams are always in text mode at the start of your program. You can change the mode of a standard stream from text to binary, without redirecting the stream, by using the `freopen` function with no file name specified. For example:

```
   fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text. You cannot change the mode of a stream to anything other than text or binary, nor can you change the file type to something other than disk. No other parameters are allowed. Note that this method is included in the SAA C definition, but not in the ANSI C standard.

Control-Z Character in Text Streams

When a text stream is connected to a character device, such as the keyboard or an OS/2 pipe, the Ctrl-Z (`\x1a`) character is treated as an end-of-file indicator, regardless of where it appears in the stream.

If Ctrl-Z is the last character in a file, it is discarded when read. Similarly, when a file ending with a Ctrl-Z character is opened in append or update mode, the Ctrl-Z is discarded. Programs compiled by the VisualAge C++ compiler do not automatically have a Ctrl-Z character appended to the end of the file when the file is closed. If you require a Ctrl-Z character at the end of your text files, you must write it out yourself.

This treatment of the Ctrl-Z character applies to text streams only. In binary streams, it is treated like any other character.

Binary Streams

A binary stream is a sequence of characters or data. The data is not altered on input or output, so the data read from a binary stream is equal to the data that was written.

If the last operation on the stream is a read operation, `fflush` discards the unread portion of the buffer. If the last operation on the stream is a write operation, `fflush` writes out the contents of the buffer. In either case, `fflush` clears the buffer.

Differences between Storing Data as a Text or Binary Stream

If two streams are opened, one as a binary stream and the other as a text stream, and the same information is written to both, the contents of the streams may differ. The following example shows two streams of different types and the hexadecimal values of the resulting files. The values show that the data is stored differently for each file.

```
#include <stdio.h>

int main(void)
{
    FILE *fp1, *fp2;
    char lineBin[15], lineTxt[15];
    int x;
```

Figure 2 (Part 1 of 2). Differences between Binary and Text Streams

```
    fp1 = fopen("script.bin","wb");
    fprintf(fp1,"hello world\n");

    fp2 = fopen("script.txt","w");
    fprintf(fp2,"hello world\n");

    fclose(fp1);
    fclose(fp2);

    fp1 = fopen("script.bin","rb");

    /* opening the text file as binary to suppress
    the conversion of internal data  */
    fp2 = fopen("script.txt","rb");

    fgets(lineBin, 15, fp1);
    fgets(lineTxt, 15, fp2);

    printf("Hex value of binary file = ");
    for (x=0; lineBin[x]; x++)
        printf("%.2x", (int)(lineBin[x]) );

    printf("\nHex value of text file   = ");
    for (x=0; lineTxt[x]; x++)
        printf("%.2x", (int)(lineTxt[x]) );

    printf("\n");

    fclose(fp1);
    fclose(fp2);

    /* The expected output is:

       Hex value of binary file = 68656c6c6f20776f726c640a
       Hex value of text file   = 68656c6c6f20776f726c640d0a  */
}
```

Figure 2 (Part 2 of 2). Differences between Binary and Text Streams

As the hexadecimal values of the file contents show in the binary stream (`script.bin`), the new-line character is converted to a line feed (`\0a`), while in the text stream (`script.txt`), the new line is converted to a carriage-return line feed (`\0d0a`).

Memory File Input/Output

When you compile with the `/Sv+` option, VisualAge C++ compiler supports files known as **memory files**. They differ from the other file types only in that they are temporary files that reside in memory. You can write to and read from a memory file just as you do with a disk file.

Using memory files can speed up the execution of your program because, under normal circumstances, there is no disk I/O when your program accesses these files. However, if your program is running in an environment where the operating system is paging, you might not get faster execution when using memory files. This loss of speed is most likely to be true if your memory files are large.

You can create a memory file in two ways:

- By specifying `type=memory` directly in your source code. For example

  ```
  stream = fopen("memfile.txt", "w, type=memory");
  ```

- By using the ddname in the `fopen` call and the `SET` command to specify the file you want your program to open.

  ```
  stream=fopen("DD:MEMFILE", "w");
  ```

  Before you run your program, use the `SET` command:

  ```
  SET DD:MEMFILE=memfile.txt, memory(y)
  ```

  The `SET DD:` statement specifies `MEMFILE` as a *data definition name* (ddname).

  Notes:

  1. You must specify the `/Sh+` compiler option to use ddnames.
  2. ddnames are not supported for use with C++ standard streams.

Once a memory file has been created, it can be accessed by the module that created it as well as by any other function within the same process. The memory file remains accessible until the file is removed by the `remove` function or until the program has terminated.

A call to `fopen` that tries to open a file with the same name as an existing memory file accesses the memory file, even if you do not specify `type=memory` in the `fopen` call.

When using `fopen` to open a memory file in write or append mode, you must ensure that the file is not already open.

Memory File I/O

Memory File Restrictions and Considerations

- You must specify the `/Sv+` option to use memory files.
- Memory files are private to the process that created them. Redirection to memory files from the command line is not supported, and they cannot be shared with any other process, including child processes. Also, memory files cannot be shared through the `system` function.
- Memory files do not undergo any conversion of the new-line character. Data is not altered on input or output.
- Memory files are unbuffered, and the `blksize` attribute is ignored. No validation is performed for the path or file name used.
- Memory file names are case sensitive. For example, the file `a.a` is not the same memory file as `A.A`:

  ```
  fopen("A.A","w,type=memory");
  remove("a.a");
  ```

  The above call to `remove` will not remove memory file `A.A` because the file name is in uppercase. Because memory files are always checked first, the function will look for memory file `a.a`, and if that file does not exist, it will remove the *disk file* `a.a` (or `A.A`, because disk files are not case sensitive).
- You can request that the temporary files created by the `tmpfile` function be either disk files or memory files. By default, `tempfile` creates disk files. To have temporary files created as memory files, set the TEMPMEM environment variable to `ON`:

  ```
  SET TEMPMEM=on
  ```

  The word `on` can be in any case. You must still specify the `/Sv+` compiler option. For more information about TEMPMEM, see Chapter 1, "Setting Runtime Environment Variables" on page 3.

Buffering

VisualAge C++ compiler uses buffers to increase the efficiency of system-level I/O. The following buffering modes are used:

Unbuffered	Characters are transmitted as soon as possible. This mode is also called unit buffered.
Line buffered	Characters are transmitted as a block when a new-line character is encountered or when the buffer is filled.
Fully buffered	Characters are transmitted as a block when the buffer is filled.

The buffering mode specifies the manner in which the buffer is flushed, if a buffer exists.

You can use the `blksize` parameter with the `fopen` function to indicate the initial size of the buffer you want to allocate for the stream. Note that you must specify the `/Sh+` compiler option to use ddnames.

If you do not specify a buffer size, the default size is 4096. Either the `setvbuf` or `setbuf` function can be used to control buffering. One of these functions may be specified for a stream. You cannot change the buffering mode after any operation on the file has occurred.

Fully buffered mode is the default unless the stream is connected to a character device, in which case it is line buffered.

To ensure data is transmitted to external storage as soon as possible, use the `setbuf` or `setvbuf` function to set the buffering mode to unbuffered.

Note: VisualAge C++ does not support pipes that are created with the `DosCreateNPipe` API.

Opening Streams Using Data Definition Names

When you specify the `/Sh+` compiler option, you can use the OS/2 `SET` command with a data definition name (ddname) as a parameter to specify the name of the file to be opened by your program. You can also use the `SET` command to specify other file characteristics.

When you use the `SET` command with ddnames, you can change the files that are accessed by each run of your program without having to alter and recompile your source code.

Notes:

1. You cannot use ddnames with the C++ standard streams.
2. The maximum number of files that can be open at any time changes with the amount of memory available.

Specifying a ddname with the SET Command

To specify a ddname, the `SET` command has the following syntax:

```
SET DD:DDNAME=filename[,option, option...]
```

where:

DDNAME	Is the ddname as specified in the source code. The ddname **must** be in uppercase.
filename	Is the name of the file that will be opened by `fopen`.

No white-space characters are allowed between the `DD` and the equal sign.

For example, you could open the file `sample.txt` in two ways:

- By putting the name of the file directly into your source code:

  ```
  FILE *stream;
  stream=fopen("sample.txt", "r");
  ```

- By using a ddname in the `fopen` call and the `SET` command to specify the file you want your program to open:

  ```
  FILE *stream;
  stream=fopen("DD:DATAFILE", "r");
  ```

  Before you run your program, use the `SET` command:

  ```
  SET DD:DATAFILE=c:\sample.txt
  ```

When the program runs, it will open the file `c:\sample.txt`. If you want the same program to use the file `c:\test.txt` the next time it runs, use the following `SET` command:

```
SET DD:DATAFILE=c:\test.txt
```

The `SET` command can be issued before your program is executed by entering it on the command line, including it in a batch file, or putting it into the `CONFIG.SYS file`. You can also use the `putenv` function from within the program to set the ddname. For example:

```
_putenv("DD:DATAFILE=sample.txt, writethru(y)");
```

For a description of `putenv`, see the *C Library Reference*.

Describing File Characteristics Using Data Definition Names

When you are defining ddnames, use the options to specify the characteristics of the file your program opens. You can specify the options in any order and in upper- or lowercase. If you specify an option more than once, only the last one takes effect. If an option is not valid, `fopen` fails and `errno` is set accordingly.

You can use the following options when specifying a ddname.

Note: The options `blksize`, `lrecl`, and `recfm` are meant to be used with record level I/O. Because record level I/O is not supported, these options are accepted but ignored.

blksize(*n* **)**
: The size in bytes of the block of data moved between the disk and the program. The maximum size is 32760 for fixed block files and 32756 for variable block files. Larger values can improve the efficiency of disk access by lowering the number of times the disk must be accessed. Typically, values below 512 increase I/O time, and values above 8KB do not show improvement.

lrecl(*n* **)**
: The size in bytes of one record (logical record length). If the value specified is larger than the value of `blksize`, the `lrecl` value is ignored.

recfm(f | v | fb | vb)[1]
: Specifies whether the record length is fixed or variable, and whether the records are stored in blocks.

f	The record size is fixed (i.e. all records are the same length) and the size of each record is specified by the `lrecl` option.
v	The record size is variable and the maximum record size is specified by the `lrecl` option.
fb	The record size is fixed and the records are stored in blocks. The record size is specified by the `lrecl` option, and the block size is specified by the `blksize` option. The block size must be an integral multiple of `lrecl`.
vb	The record size is variable and the records are stored in blocks. The maximum record size is specified by the `lrecl` option, and the block size is specified by the `blksize` option.

share (read | none | all)
: Specifies the file sharing.

read	The file can be shared for read access. Other processes can read from the file, but not write to it.
none	The file cannot be shared. No other process can get access to the file (exclusive access).
all	Allows the file to be shared for both read and write access. Other processes can both read from and write to the file.

1 The default values for these options are underlined.

writethru(n | y)
Determines whether to force the writing of OS/2 buffers.

n	Turns off forced writes to the file. The system is not forced to write the internal buffer to permanent storage before control is returned to the application.
y	Forces the system to write to permanent storage before control is returned to the application. The directory is updated after every write operation. Use `writethru(y)` if data must be written to the disk before your program continues. This can help make data recovery easier should a program interruption occur.

Note: When `writethru(y)` is specified, file output will be noticeably slower.

memory(n | y)
Specifies whether a file will exist in permanent storage or in memory.

n	Specifies that the file will exist in permanent storage.
y	Specifies that the file will exist only in memory. The system uses only the OS/2 file name. All other parameters, such as a path, are ignored. You must specify the `/Sv+` option to enable memory files.

fopen Defaults

A call to `fopen` has the following defaults:

share(read)	The file can be shared for read access. Other processes can read from the file, but not write to it.
writethru(n)	The file is opened with no forced writes to permanent storage.

Full buffering is used unless the stream is connected to a character device, then it it is line buffered.

For more information on `fopen`, refer to the *C Library Reference*.

Precedence of File Characteristics

You can describe your data both within the program, by `fopen`, and outside it, by ddname, but you do not always need to do so. There are advantages to describing the characteristics of your data in only one place.

Opening a file by ddname may require the merging of the information internal and external to the program. In the case of a conflict, the characteristics described by using `fopen` override those described using a ddname. For example, given the following ddname statement and `fopen` command:

```
SET DD:ROGER=danny.c, memory(n)
stream = fopen("DD:ROGER", "w, type=memory")
```

the file `danny.c` will be opened as a memory file.

Closing Files

The `fclose` function is used to close a file. On normal program termination, the compiler library routines automatically close all files and flush all buffers. When a program ends abnormally, all files are closed but the buffers are not flushed.

Input/Output Restrictions

The following restrictions apply to input/output operations:

- Seeking within character devices and OS/2 piped files is not allowed.
- Seeking past the end of the file is not allowed for text files. For binary files that are opened using any of `w, w+, wb+, w+b`, or `wb`, a seek past the end of the file will result in a new end-of-file position and nulls will be written between the old end-of-file position and the new one.

Note: When you open a file in append mode, the file pointer is positioned at the end of the file.

I/O Considerations When You Use Presentation Manager

Standard I/O functions such as `printf` write to OS/2 file handle 1, which is the default destination of **stdout** and **cout**. Unless you redirect the output and messages, you cannot see them through the Presentation Manager (PM) interface.

There are two ways to display the output sent to **stdout** or **cout** depending on whether you want to see the output while the program is running or after it has finished:

1. To see the output while the program is running, you must pipe the output stream to some other program that reads input and displays it using PM calls. For example, suppose you had a program called "display" which used PM calls to write to the screen. You could pipe the output from `junko.exe` to the program `display` using the following command:

   ```
   junko | display
   ```

2. To view the output after the program has finished, redirect the output stream to a file. You can do this from a command line, for example:

   ```
   junko > file.out
   ```

   or from within the file using the `freopen` function:

   ```
   freopen("file.out", "w", stdout);
   ```

To send output from a VisualAge C++ application directly to a PM window, you must use PM calls.

All error messages during run time go to OS/2 file handle 2, which is the default destination of **stderr**, **cerr**, and **clog**. Like output to file handle 1, these messages are not visible through the PM interface. To see the error messages, you must redirect the error stream to a file.

C++ programs using User Interface classes can force the output messages of C++ exceptions in the library to either **stderr**, **stdout**, or to a queue. To do this, prior to running a Presentation Manager application, two environment variables, `ICLUI TRACE` and `ICLUI TRACETO` must be set.

`ICLUI TRACE` can be set to `off` (the default), `on`, or `noprefix`. This last value means the trace is set on but no prefix information is written to trace.

`ICLUI TRACETO` can then be set to **stdout**, **stderr**, or `queue` (the default). The `queue` value causes the trace to be written to a 32-bit OS/2 queue named `\\QUEUES\PRINTF32`.

For more details on redirecting output, see “Redirecting Standard Streams” on page 15.

4 Optimizing Your Program

Two aspects of your program can be optimized. You can decrease the size of your program or you can improve your program's execution performance. Note that in some cases, optimizing for size may result in slower programs, and optimizing for speed may result in larger programs. In addition, when you optimize your code you may uncover bugs that had not been evident before.

This chapter provides guidelines only. To obtain the best results for either performance or module size, experiment with the techniques suggested.

Standard Optimization Considerations

It is assumed you have already implemented the obvious program changes which typically yield the initial, dramatic, performance improvements. Program changes you should already have considered include: choosing efficient algorithms with small memory footprints; avoiding duplicate copies of data; and passing by value versus passing by reference wherever possible. While not a technical pre-requisite to the fine-tuning methods discussed here, if you have not already examined your program for these types of improvements, we recommend you do so before continuing with this chapter.

For help on determining the execution profile of your program, see the discussion of the Performance Analyzer in the *User's Guide*. The benefits to your program will vary depending on your code and on the opportunities for optimization available to the compiler.

Fine-tuning Techniques for Optimizing Code

This chapter describes fine-tuning techniques which have the potential to squeeze that final percentage point or two of performance improvement out of your program, for those situations and applications where peak efficiency is required. Because the size of your program affects both the load time and the runtime characteristics of your application, it is best to do size tuning before performance tuning. We have presented the topics in that order.

Reducing Program Size

This section lists the methods you can use to decrease the size of your executable module.

Reducing Program Size

Coding Techniques

The following list describes relatively quick and simple ways you can make your modules smaller:

- Use **#pragma strings(readonly)** to make your strings read-only. In C++ programs, strings are read-only by default.
- Use the `\exepack:1` option when linking. This option allows you to build smaller, compressed executables. The time taken to expand the executable, which is done automatically by the OS/2 Application Loader when the program is run, is less than the I/O time of bringing in the larger uncompressed file.

  Note: OS/2 Version 3.0 allows you to specify either `\exepack:1` or `\exepack:2` for this linker option. The `\exepack:2` version uses an improved compression algorithm but is not compatible with OS/2 Versions 2.0, 2.1 or 2.11. In those cases, use `\exepack:1`.

  Note: Neither `exepack` option would be used with the `align` directive is also used with a parameter less than 512.
- When you declare or define structures or C++ classes, take into account the alignment of data types. Declare the largest members first to reduce wasted space between members.
- If you do not use the intermediate code linker, arrange your own external data to minimize gaps in alignment.
- Avoid assigning structures.
- Avoid assigning structures if your structures are large or if you use **#include <string.h>**. Instead, use `memcpy` to copy the structure.
- If you do not use the `argc` and `argv` arguments to **main**, create a dummy `_setuparg` function that contains no code.

Using Libraries and Library Functions

Your choice of libraries and of library functions affects the size of your code. The guidelines below can add up to a greater reduction in size than those listed above, but they can also require more effort on your part. In some cases, they require you to write your own code for such things as buffering or exception handling.

- Use the subsystem library whenever possible. This library has no runtime environment, so the initialization, termination, and exception handling code is not included. It also includes fewer library functions than the standard library.
- Use the low-level I/O functions. Note that you must provide your own buffering for these functions.

- Disable the inlining of intrinsic C functions.

  The /Oc switch disables expansion of intrinsic fuctions whenever the function call is smaller than the inlined function. You must specify /O whenever you wish to use the /Oc switch.

  Certain string manipulation, floating-point, and trigonometric functions are inlined by default. (See “Intrinsic Functions” on page 357 for a list of these functions.) To selectively disable the inlining, parenthesize the function call, for example:

  ```
  (strlen)(x);
  ```

  For most of the floating-point built-in functions, this recommendation does not apply because the inlined code is probably smaller than a generated call instruction.

Choosing Compiler Options

The following list names the compiler options to use to make your executable module smaller. Unless noted, these options are not set by default.

`/Gd+`	Links dynamically to the runtime library. If you link statically, code for all the runtime functions you call is included in your executable module.
`/Gf+`	Generates code for fast floating-point execution and eliminates certain conversions.
`/Gh-`	Does not generate execution trace and analyzer hooks which would increase module size. This is the default.
`/Gi+`	Generates code for fast integer execution and eliminates certain conversions.
`/Gv-`	Does not save and restore the DS and ES registers for external function calls. This is the default.
`/Gw-`	Does not generate an `FWAIT` instruction after each floating-point load instruction. This is the default.
`/Gx+`	For C++ programs only, suppresses generation of exception handling code.
`/G3`	Optimizes for the 386 processor. This is the default. Optimizing for the 486 or Pentium microprocessor generates extra code. Code compiled with `/G3` runs on a 486 or Pentium microprocessor.
`/O+`	Turns on optimization.
`/Oc+`	Turns on optimization for size. You must also specify /O.
`/Oi-`	Does not inline user functions. Inlining reduces overhead but increases module size. When `/O-` is specified, this is the default. When `/O+` is specified, `/Oi+` becomes the default.

`/Ol+`	Passes code through the intermediate code linker. The intermediate linker removes unused variables and sorts external data to provide maximal packing. For best results, use the `/Gu+` option to specify that defined data is not used by external functions. See the *User's Guide* for more information about the intermediate linker.
`/Sh-`	Does not include ddname support. This is the default.
`/Sv-`	Does not include memory file support in the library. This is the default.
`/Ti-`	Does not generate debug or Performance Analyzer information, which would increase module size. This is the default.
`/Tx-`	Provides only the exception message and address when an exception occurs instead of a complete machine-state dump. This is the default.

If you link your program in a separate link step, specify the `/ALIGN:1` linker option to align segments on 1-byte boundaries. The default alignment is 512 byte boundaries. Alternatively, you could specify the `/EXEPACK` linker option, which compresses repeated byte patterns within pages of data. Neither of these linker options reduces the memory image size, only the disk size.

Note: Do not use `/ALIGN:1` and `/EXEPACK` together as doing so can significantly lower performance.

Improving Program Performance

This section lists the methods you can use to improve the speed of your program.

Choosing Libraries

Your choice of runtime libraries can affect the performance of your code:

- Use the subsystem library whenever possible. Because there is no runtime environment for this library, its load and initialization times are faster than for the other libraries.
- Use the single-thread library for single-thread programs. The multithread library involves extra overhead.
- If your application has multiple executable modules and DLLs, create and use a common version of a runtime library DLL. See "Creating Your Own Runtime Library DLLs" on page 83 for more information.

Allocating and Managing Memory

The following list describes ways to improve performance through better memory allocation and management:

- If you allocate a lot of dynamic storage for a specific function, use the `_alloca` function. Because `_alloca` allocates from the stack instead of the heap, the storage is automatically freed when the function ends. In some cases however, using `_alloca` can detract from performance. It causes the function that calls it to chain the EBP register, which creates more code in the function prolog and also eliminates EBP from use as a general-purpose register. If you are not allocating much dynamic storage, this overhead can outweigh the benefits of using `_alloca`. For this reason, if your function does not allocate a lot of dynamic storage, use other memory allocation functions.

- You can use `malloc`, `DosAllocMem`, or if programming in C++, **new** to allocate storage. In general, `DosAllocMem` is faster, but you must do your own heap management and you cannot use `realloc` to reallocate the memory. However, `malloc` manages the heap for you and the storage it returns can be reallocated with `realloc`. In addition, `malloc` is portable, while `DosAllocMem` is not. When programming in C++, use **new**. **new** sets up virtual function tables (VFTs) which `malloc` and `DosAllocMem` do not. It also provides additional type checking not available when using `malloc` or `DosAllocMem`.
- When you use `malloc`, the amount of storage allocated is actually the amount you specify plus a minimal overhead that is used internally by the memory allocation functions.
- When you copy data into storage allocated by `calloc`, `malloc`, or `realloc`, copy it to the same boundaries on which the compiler would align them. In particular, aligning double precision floating-point variables and arrays on 8-byte boundaries can greatly improve performance on the 486 and Pentium microprocessors. For more information about the mapping of data, see “Data Mapping” on page 399.
- When you declare or define structures or C++ classes, take into account the alignment of data types. Declare the largest members first to reduce wasted space between members and to reduce the number of boundaries the compiler must cross. The alignment is especially important if you pack your structure or class.
- After freeing or reallocating storage, periodically call `_heapmin` to release the unused storage to the operating system and reduce the working set of your program. A reduced working set causes less swapping of memory to disk, resulting in better performance. Experiment to determine how often you should call `_heapmin`.

Using Strings and String Manipulation Functions

The handling of string operations can affect the performance of your program:

- In C, use **`#pragma strings (readonly)`** to make your strings read-only. In C++, strings are read-only by default. Using the pragma also causes the compiler to put out only copy of strings that are used in more than one place.

  If you use the intrinsic string functions, the compiler can better optimize them if it knows that any string literals they are operating on will not be changed.
- When you store strings into storage allocated by `malloc`, align the start of the string on a doubleword boundary. This alignment allows the best performance of

the string functions. The compiler performs this alignment for all strings it allocates.

- Keep track of the length of your strings. If you know the length of your string, you can use `memcpy` instead of `strcpy`. The `memcpy` function is faster because it does not have to search for the end of the string.
- Avoid using `strtok`. Because this function is very general, you can probably write a function more specific to your application and get better performance.

Performing Input and Output

There are a number of ways to improve your program's performance of input and output:

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as `open` and `close`. These functions are faster and more specific to the application than the stream I/O functions like `fopen` and `fclose`. You must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page. Because `malloc` uses extra storage as overhead, allocating storage in a multiple of the page size actually results in more pages being allocated than required. Instead, use `DosAllocMem` to allocate this storage for the buffer.
- If you know you have to process an entire file, the following technique has the advantage of reducing disk I/O, provided the file is not so big that excessive swapping will occur. Determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using `DosRead`, and then process the data in the buffer.
- If you perform frequent read or write operations on your temporary files, create them as memory files. I/O operations can be performed more quickly on memory files than on disk files. To use memory files, you must specify the `/Sv+` option.
- Instead of `scanf` and `fscanf`, use `fgets` to read in a string, and then use one of `atoi`, `atol`, `atof`, or `_atold` to convert it to the appropriate format.
- Use `sprintf` only for complex formatting. For simpler formatting, such as string concatenation, use a more specific string function.
- When reading input, read in a whole line at once rather than one character at a time.

Designing and Calling Functions

Whether you are writing a function or calling a library function, there are a few things you should keep in mind:

- Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required and the compiler never needs to emit eyecatcher instructions for the function (see "Eyecatchers" on page 152).
- When designing a function, place the most used parameters in the left most position in the function prototype. The left most parameters have a better chance of being stored in a register.
- Avoid passing structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value, a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union.
- If you call another function near the end of your function and pass it the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then reuse the storage that the parameters are in and does not have to generate code to reorder them.
- Use the intrinsic and built-in functions, which include string manipulation, floating-point, and trigonometric functions. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization. Your functions are automatically mapped to intrinsic functions if you include the VisualAge C++ header file, however, in C only, this mapping is overridden if you `#undef` the macro name.
- Be careful when using intrinsic functions in loops. Many intrinsic functions use multiple registers. Some of the registers are specific and cannot be changed. In the loop, the number of values to be placed in registers increases while the number of registers is limited. As a result, temporary values such as loop induction variables and results of intermediate calculations often cannot be stored in registers, thus slowing your program performance.

  In general, you will encounter this problem with the intrinsic string functions rather than the floating-point functions.
- Use recursion only where necessary. Because recursion involves building a stack frame, an iterative solution is always faster than a recursive one.

Other Coding Techniques

The following list describes other techniques you can use to improve performance:

- Minimize the use of external (`extern`) variables to reduce aliasing and so improve optimization.

Improving Program Performance

- Avoid taking the address of local variables. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.
- Avoid using `short int` values, except in aggregates. Because all integer arithmetic is done on `long` values, using `short` values causes extra conversions to be performed.
- If you do division or modulo arithmetic by a divisor that is a power of 2, if possible, make the dividend unsigned to produce better code.
- Use **`#pragma alloc_txt`** and **`#pragma data_seg`** to group code and data respectively, to improve the locality of reference. Using **`#pragma alloc_txt`** causes functions that are used at the same time to be stored together. They might then fit on a single page that can be used and then discarded. You can use Performance Analyzer to determine which functions to group together. **`#pragma data_seg`** works in a similar manner for grouping data.
- Use **_Optlink** linkage wherever possible. Keep **_Optlink** as your default linkage and use linkage keywords to change the linkage for specific functions.
- If a loop body has a constant number of iterations, use constants in the loop condition to improve optimization. For example, the statement `for (i=0; i<4; i++)` can be better optimized than `for (i=0; i<x; i++)`.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Use the intermediate code linker to improve optimization. See the *User's Guide* for information about the intermediate linker.
- Inline your functions selectively. Inlined functions require less overhead and are generally faster than a function call. The best candidates for inlining are small functions that are called frequently. Large functions and functions that are called rarely may not be good candidates for inlining.

  For best results, use the Performance Analyzer to decide which functions you should inline and qualify the **_Inline** keyword (or **`inline`** for C++ files). For a discussion of using Performance Analyzer in this manner, see the *User's Guide*. (Using automatic inlining, specifying `/Oi` with a value, is not as effective.) Using the intermediate code linker with user inlining can improve your program performance even more.

Some coding practices, although often necessary, will slow down program performance:

- Calling 16-bit code. The compiler performs a number of conversions to allow interaction between 32-bit and 16-bit code.
- Using the `setjmp` and `longjmp` functions. These functions involve storing and restoring the state of the thread.
- Using **`#pragma handler`**. This **#pragma** causes code to be generated to register and deregister an exception handler for a function.
- Using unprototyped variable argument functions. Because of the nature of the **_Optlink** calling convention, unprototyped variable-length argument lists make performance slower. Prototype all of your functions. Also, use the **_System** calling convention for any variable argument functions.

C++-Specific Considerations

The following performance hints apply only to C++ programs:

- Because C++ objects are often allocated from the heap and have a limited scope, memory usage in C++ programs affects performance more than in C programs. To improve memory usage and performance:
  - Tailor your own **new** and **delete** operators.
  - Allocate memory for a class before it is required.
  - Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an object manager. Each time you create an instance of an object, you pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
  - Avoid copying large complex objects.
- When you use the Collection classes from the *Open Class Libary* to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can result in improved performance with minimal code change.
- Use virtual functions only when they are necessary. They are usually compiled to be indirect calls, which are slower than direct calls.
- Use `try` blocks for exception handling only when necessary because they can inhibit optimization. Use the `/Gx+` option to suppress the generation of exception handling code in programs where it is not needed. Unless you specify this option, some exception handling code is generated even for programs that do not use `catch` or `try` blocks.

- Avoid using overloaded operators to perform arithmetic operations on user-defined types. The compiler cannot perform the same optimizations for objects as it can for simple types.
- Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on. A simple assignment using an overloaded operator can generate many lines of code.
- When you define structures or data members within a class, define the largest data types first to align them on the doubleword boundary.
- Usually, you should not declare virtual functions inline. If all virtual functions in a class are inline, the virtual function table of that class and all the virtual function bodies will be replicated in each compilation unit that uses the class.

Choosing Compiler Options

The following list names the compiler options that can improve performance and describes the action of each option.

Note: Of these options, only `/Om-` is a default.

Option	**Effect**
`/Gf+`	Generates code for fast floating-point operations.
`/Gi+`	Generates code for fast integer operations.
`/Gx+`	For C++ programs only, suppresses generation of exception handling code.
`/G[3|4|5]`	Optimize for the 386 (`/G3`), 486 (`/G4`), or Pentium (`/G5`) microprocessor. Use the appropriate option for the processor you are using or plan to use. If you do not know what processor your application will run on, use the `/G3` option.
`/O+`	Turns on optimization for speed. Specifying `/O+` also causes `/Op+` (enable optimizations involving the stack pointer), `/Os+` (invoke the instruction scheduler), and `/Oi+` (inline user functions) to be specified.
`/Oi+`	Inlines user functions.
`/Ol+`	Passes code through the intermediate code linker. Using the intermediate linker can result in better optimized code. For best results, use the `/Gu+` option also to specify that data that is defined in the `.DLL` or `.EXE` being built is not used by external functions. See the *User's Guide* for more information about the intermediate linker.
`/Om-`	Does not limit the working set size of the compiler so that the compiler can inline more user code.

The following options improve the performance of your code by preventing the generation of objects or information that can degrade performance. Note that these are set by default:

Option	**Effect**
`/Gh-`	Prevents the generation of execution trace and analyzer hooks.
`/Gr-`	Generates code to run in the usual operating system environment. If you use `/Gr+`, the code generated runs at ring `0`, and the performance suffers. Some code, such as device drivers, must run at ring `0`.
`/Gv-`	Does not save and restore the DS and ES registers for external function calls.
`/Gw-`	Prevents the generation of `FWAIT` instruction after each floating-point load instruction.
`/Ti-`	Does not generate debug information.

If your program has only one thread, use the `/Gs+` option to disable stack probes. (`/Gs-` is the default.) Because the stack of the first thread is always fully committed, stack probes are not necessary in single-thread programs. If your program has multiple threads, stack probes serve a useful purpose and you should probably use them. 🕮 See the *User's Guide* for more information about stack probes.

If you link your executable files in a separate link step, specify the `/BASE:65536` linker option to tell the linker your executable file will be loaded at 64K. The linker can then resolve a number of references that would otherwise have to be resolved by the loader at load time and by the pager as the program runs. When you use `icc` to link your program, it specifies this option for you by default.

Note: Do not use the `/BASE:65536` for DLLs.

Specifying Linker Options

Using the following linker options can lead to improved performance. When `icc` invokes the linker, it passes these options by default:

`/BASE:65536` Specify the starting address of the program. Because the OS/2 operating system always loads executable programs at 64K, you can give the linker the address 65536 (or `0x10000`). If the linker knows where the program will be loaded, it can resolve relocation information at link time, resulting in a smaller and faster executable module.

Note: If you use the `/BASE` option to compile DLLs, you may find that there is no performance improvement. DLLs load more slowly if the starting address that is specified for them is not their load address. In addition, if the specified starting address for a DLL is too low, the size of the private address space is reduced and there may not be much heap available for you to use.

Improving Program Performance

`/EXEPACK` or `/EXEPACK[1|2]` Pack the `.EXE` or `.DLL` file. The size of the module is reduced, thereby reducing load time. `EXEPACK` and `/EXEPACK:1` are equivalent, `/EXEPACK:2` uses a more effective compression algorithm but is available only under OS/2 Warp Version 3.0.

5 Creating Multithread Programs

This chapter describes how to use the VisualAge C++ compiler to create multithread programs and discusses restrictions of the multithread environment. It also describes the sample multithread program that you may have installed which is included with the VisualAge C++ product. For instructions on how to compile and run the sample program, see “Sample Multithread Program” on page 59.

Multithread programming is a feature of the OS/2 operating system and is supported by the VisualAge C++ compiler with:

- Code generation and linking options. (See “Compiling and Linking Multithread Programs” on page 59 for more information.)
- Multithread libraries. (See “Using the Multithread Libraries” on page 49 for more information.)

No multithread support is available in the subsystem libraries.

What Is a Multithread Program?

A multithread program is one whose functions are divided among several threads. A *process* is an executing application and the resources it uses, a *thread* is the smallest unit of execution within a process. Other than its stack and registers, a thread owns no resources; it uses those of its parent process. This chapter discusses threads, references to processes are for contrast only.

Multithread programs allow more complex processing than single-thread programs. In a single-thread program, all operations are performed serially. That is, one operation begins when the preceding one has finished.

The advantage of having multiple threads are:

1. on multi-processor systems, that threads can execute concurrently. and thus the entire multithread program is completed faster.
2. on single or multiple processor systems, that if any thread is blocked (waiting for I/O to complete) then the rest of your application can continue to process or respond to user data.

Multithread Programs

Although threads within a process share the same address space and files, each thread runs as an independent entity and is not affected by the control flow of any other thread in the process. Because a function from any thread can perform any task, such as input or output, threads are well suited to programs that have multiple uses of the same data or resources.

Thread Control

There are three mechanisms for create and deleting threads under the VisualAge C++:

- `_beginthread` and `_endthread` from the multithread libraries,
- `DosCreateThread` and `DosExit` from the OS/2 API, or
- `IThread` from the User Interface classes.

These are discussed in more detail below.

_beginthread and **_endthread**

The multithread libraries provide two functions, `_beginthread` and `_endthread`, to create new threads and to end them. You should use `_beginthread` to create any threads that call VisualAge C++ library functions. When the thread is started, the library environment performs certain initializations that ensure resources and data are handled correctly between threads.

The VisualAge C++ compiler does not limit the number of threads you can create, but the OS/2 operating system does. The VisualAge C++ product also provides the global variable `_threadid` that identifies your current thread, and the function `_threadstore` that gives you a private thread pointer to which you can assign any thread-specific data structure.

For more information on the number of threads allowed, see the online *OS/2 Programming Reference*. For more detail on the functions `_endthread` and `_beginthread`, see the *C Library Reference*.

DosCreateThread and **DosExit**

You can also create threads with the `DosCreateThread` API. The function that is to run on the thread created by `DosCreateThread` must have **_System** linkage. If you need to start a new thread for a function with any other type of linkage, you cannot use `DosCreateThread`.

Threads created by the `DosCreateThread` API do not have access to the resource management facilities or to VisualAge C++ exception handling, you must use a **`#pragma handler`** directive for the thread function to ensure correct exception handling. You should also call `_fpreset` from the new thread to ensure the floating-point control word is set correctly for the thread. Although you can use `DosExit` to end threads created with `DosCreateThread`, you should use `_endthread` to ensure that the necessary cleanup of the environment is done.

IThread

The `start` member function of the `IThread` class is used to start additional threads. This member function has three overloaded versions and three corresponding constructors for:

- Functions compatible with `_beginthread`. That is, the functions which have OS/2 linkage, take one argument of `void*` type, and return `void`.
- Functions compatible with `DosCreateThread`. That is, the functions which have **_System** linkage, take one argument of `unsigned long` type, and return `void`.
- Any other function.

Because `IThread` can handle functions which fall under both of the previously discussed thread control mechanisms, as well as being able to handle functions which fall under neither, it is the preferred thread handling mechanism. It does not only what other mechanisms do, it does it better. Unlike `_beginthread`, you don't have to explicitly call `_endthread` to clean up the environment, and unlike `DosCreateThread`, you don't have to write your own exception handler.

You can use the `IThread` class in your multithread programs to :

- Set thread priority
- Set thread attributes
- Do a reference count for objects dispatched on a thread so they are automatically deleted when the thread ends
- Dispatch a member function of a C++ object on a separate thread
- Control other aspects of your threads.

For a description of the `IThread` class and how to use it, see the *Open Class Library Reference*.

Using the Multithread Libraries

VisualAge C++ compiler has two standard libraries that provide library functions for use in multithread programs. The `CPPOM30.LIB` library is a statically linked multithread library, and `CPPOM30I.LIB` is an import multithread library, with the addresses of the functions contained in VisualAge C++ DLLs.

In addition to the above two standard libraries, the User Interface Class library is also available in multithread form. A singlethread version of this library is not provided.

Not all of the VisualAge C++ Standard class libraries are available for multithread programs. The Complex Mathematics library is available for both single- and multithread programs. The single-thread Complex library is COMPLEX.LIB, while the multithread version is COMPLEXM.LIB. The C++ I/O Stream library is built into both the VisualAge C++ single-thread and the multithread runtime libraries. The User Interface class library also offers an `IThread` class that is an encapsulation of the OS/2 APIs for multithread programming.

When you use the multithread libraries, you have more to consider than with the single-thread libraries. For example, because many library functions share data and other resources, the access to these resources must be serialized (limited to one thread at a time) to prevent functions from interfering with each other. Other functions can affect all threads running within a process. Global variables and error handling are also affected by the multithread environment.

Reentrant Functions

Reentrant functions are those which can be suspended at any point and reentered, after which they can return to that same point to resume processing, with no adverse effects. Because these functions use only local variables, they cannot interfere with each other. Access to these functions is not serialized.

All functions in the C++ Complex Mathematics Library are fully reentrant. The I/O Stream Library functions are nonreentrant.

The following functions are reentrant:

absolut	fstat	localtime	stat	strupr
acos	_ftime	log	strcat	strxfrm
asctime	_fullpath	log10	strchr	swab
asin	gamma	_lrotl	strcmp	tan
assert	_gcvt	_lrotr	strcmpi	tanh
atan	_getcwd	lsearch	strcoll	time
atan2	_getdcwd	_ltoa	strcpy	_toascii
atof	_getdrive	_makepath	strcspn	tolower
atoi	getpid	mblen	_strdate	_tolower
atol	gmtime	mbstowcs	strerror	toupper
atold	hypot	mbtowc	_strerror	_toupper
bsearch	isalnum	memccpy	strftime	_tzset
_cabs	isalpha	memchr	stricmp	_ultoa
ceil	isascii	memcmp	strlen	utime
chdir	iscntrl	memcpy	strlwr	vsprintf
_chdrive	isdigit	memicmp	strncat	wait

clock	isgraph	memmove	strncmp	wcscat
cos	islower	memset	strncpy	wcschr
cosh	isprint	mkdir	strnicmp	wcscmp
ctime	ispunct	mktime	strnset	wcscpy
_cwait	isspace	modf	strpbrk	wcscspn
difftime	isupper	pow	strrchr	wcslen
div	isxdigit	qsort	strrev	wcsncat
_ecvt	_itoa	rmdir	strset	wcsncmp
erf	_j0	_rotl	strspn	wcsncpy
erfc	_j1	_rotr	strstr	wcspbrk
exp	_jn	sin	_strtime	wcsrchr
fabs	labs	sinh	strtok	wcsspn
_fcvt	ldexp	_splitpath	strtod	wcstombs
floor	ldiv	sprintf	strtol	wctomb
fmod	lfind	sqrt	strtold	_y0
_freemod	_loadmod	sscanf	strtoul	_y1
frexp				_yn

Although the reentrant functions do not require serialization of data access, there is an important exception: if you pass a pointer as a parameter, the function may no longer be reentrant and may therefore require that access is serialized. In short, if you try to use the same piece of memory (such as the memory pointed to by a pointer parameter) from multiple threads, the results are unpredictable.

Nonreentrant Functions

Other nonreentrant library functions can access data or resources that are common to all threads in the process, including files, environment variables, and I/O resources. To prevent any interference among themselves, these functions use *semaphores*, provided by the OS/2 operating system, to serialize access to data and resources. Semaphores are described in detail in the online *Control Program Guide and Reference*.

Operations involving file handles and standard I/O streams are serialized so that multiple threads can send output to the same stream without intermingling the output.

Nonreentrant Functions

Example of Serialized I/O

If thread1 and thread2 execute the calls in the example below, the output could appear in several different ways, but never garbled as shown at the end of the example.

```
#include <stdio.h>

int done_1 = 0;
int done_2 = 0;

void _Optlink thread1(void)
{
   fprintf(stderr,"This is thread 1\n");
   fprintf(stderr,"More from 1\n");
   done_1 = 1;
}

void _Optlink thread2(void)
{
   fprintf(stderr,"This is thread 2\n");
   fprintf(stderr,"More from 2\n");
   done_2 = 1;
}

int main(void)
{
   _beginthread(thread1, NULL, 4096, NULL);
   _beginthread(thread2, NULL, 4096, NULL);

   while (1)
   {
      if (done_1 && done_2)
         break;
   }
   return 0;
}
```

Figure 3 (Part 1 of 2). Example of Serialized I/O

```
/* Possible output could be:

        This is thread 1
        This is thread 2
        More from 1
        More from 2
or
        This is thread 1
        More from 1
        This is thread 2
        More from 2
or
        This is thread 1
        This is thread 2
        More from 2
        More from 1

   The output will never look like this:

        This is This is thrthread 1
        ead 2
        More froMore m 2
        from 1                          */
```

Figure 3 (Part 2 of 2). Example of Serialized I/O

Several nonreentrant functions have specific restrictions:

- The getc, getchar, putc, and putchar file I/O operations are implemented as macros in the single-thread C libraries. In the multithread libraries, they are redefined as functions to implement any necessary serialization of resources.
- Use the _fcloseall function only after all file I/O has been completed.
- When you use printf or vprintf and the subsystem libraries, you must provide the necessary serialization for **stdout** yourself.

The functions in the C++ I/O Stream Library are also nonreentrant. To use these I/O Stream objects in a multithread environment, you must provide your own serialization either using the OS/2 semaphore APIs or the IResourceLock, IPrivateResource, and ISharedResource classes from the User Interface classes.

Global Variables in Multithread Programs

Process Control Functions

The process termination functions `abort`, `exit`, and `_exit` end all threads within the process, not just the thread that calls the termination function. In general, you should allow only thread 1 to terminate a process, and only after all other threads have ended.

Notes:

1. If your program exits from a signal or exception handler it may be necessary to terminate the process from a thread other than thread 1.
2. A routine that resides in a DLL must **not** terminate the process, except in the case of a critical error. If the DLL and the executable for the process have different runtime libraries, terminating the process from the DLL would bypass any `onexit` or `atexit` functions that the executable may have registered.

Signal Handling in Multithread Programs

Signal handling, as described in Chapter 14, "Signal and OS/2 Exception Handling" on page 217, also applies to the multithread environment. The default handling of signals is usually either to terminate the program or to ignore the signal. Special-purpose signal handling, however, can be complicated in the multithread environment.

Signal handlers are registered independently on each thread. For example, if thread 1 calls `signal` as follows:

```
signal(SIGFPE, handlerfunc);
```

then the handler `handlerfunc` is registered for thread 1 only. Any other threads are handled using the defaults.

A signal is always handled on the thread that generated it, except for SIGBREAK, SIGINT, and SIGTERM. These three signals are handled on the thread that generated them only if they were raised using the `raise` function. If they were raised by an exception, they will be handled on thread 1.

For more information and examples on handling signals, refer to Chapter 14, "Signal and OS/2 Exception Handling" on page 217.

Global Data and Variables

The following two variables need to have a unique value for each thread in which they are defined:

- `errno`
- `_doserrno`

When a thread defines either of these two variables, it automatically gets its own value for them that is not affected by the values that the variables might have in other threads. Other variables, such as `_environ` are common across all threads and do not automatically get unique values in each thread that uses them.

For example, the following program shows how the value of `errno` is unique to each thread. Although an error occurs in the thread `openProc`, the value of `errno` is `0` because it is checked from the `main` thread.

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

int done = 0;

void _Optlink openProc(void * argument)
{
   FILE *filePointer ;
   filePointer = fopen("C:\\OS2","w");
   printf("openProc, errno = %d\n",errno);
   done = 1;
}

int main(void)
{
   char holder[80];

   errno = 0 ;
   _beginthread(openProc,NULL,4096,NULL) ;
```

Figure 4 (Part 1 of 2). Example of a Per-Thread Variable

```
   while (1)  /* Break only when the thread is done. */
   {
      printf("Press <enter> to continue.\n");
      gets(holder);
      if (done)
         break ;
      printf("The thread is still executing! \n");
   }

   printf("Main program, errno = %d.\n",errno);
   return 0;

   /* The expected output is:

      Press <enter> to continue.
      openProc, errno = 60

      Main program, errno = 0.     */

}
```

Figure 4 (Part 2 of 2). Example of a Per-Thread Variable

When you call `longjmp`, the buffer you pass to it must have been initialized by a call to `setjmp` on the same thread. If the buffer was not initialized on the same thread, the process terminates.

The internal buffers used by `asctime`, `ctime`, `gmtime`, and `localtime` are also allocated on a per-thread basis. That is, these functions return addresses of buffers that are specific to the thread from where the function was called.

There is one seed per thread for generating random numbers with the `rand` and `srand` functions. This ensures that the pseudorandom numbers generated in each thread are independent of other threads. Each thread starts with the same seed (1); that is, each thread gets the same sequence of pseudorandom numbers unless the seed is changed by a call to `srand`.

Global Variables Requiring Serialization

These global variables containing environment strings should be treated as read-only data. They should only be modified by library functions:

```
int _daylight;
long _timezone;
char *_tzname[2];

char _osmajor;
char _osminor;
char _osmode;

char **_environ;
```

Note: The `_timezone` variable contains the time difference (in seconds) between the local time and Greenwich Mean Time (GMT).

The environment strings are copied from the OS/2 environment when a program starts. This procedure is the same in multithread and single thread programs. Because all threads share the environment strings, any change made to the strings by one thread affects the environment accessed by the other threads.

to set the environment variables. Each thread can call `getenv` to obtain a copy of the environment strings and copy the string to a private data buffer so that any later changes to the environment by `putenv` will not affect it. If the thread must always access the latest version of the environment strings, it must call `getenv` each time. 📖 The `putenv` and `getenv` functions are described in the *C Library Reference*.

Using Common Variables

User variables that are referenced by multiple threads should have the attribute `volatile` to ensure that all changes to the value of the variable are performed immediately by the compiler. For example, because of the way the compiler optimizes code, the following example may not work as intended when compiled with the `/O+` option:

Global Variables in Multithread Programs

```
static int common_var;

/*  code executing in thread 1  */

   common_var = 0;
       ...
   common_var = 1;
       ...
   common_var = 2;

/*  code executing in thread 2  */

   switch (common_var)
   {
      case 0:
          ...
        break;
      case 1:
          ...
        break;
      default:
          ...
        break;
    }
```

When optimizing, the compiler may not immediately store the value 1 for the variable `common_var` in thread 1. If it determines that `common_var` is not accessed by this code until after the value 2 is stored, it may never store the value 1. Thread 2 therefore does not necessarily access the true value of `common_var`.

Declaring a variable as `volatile` indicates to the compiler that references to the variable have side effects, or that the variable may change in ways the compiler cannot determine. Optimization will not eliminate any action involving the `volatile` variable, and changes to the value of the variable are then stored immediately.

Compiling and Linking Multithread Programs

When you compile your multithread program, you must specify that you want to use the multithread libraries described in “Using the Multithread Libraries” on page 49. Because threads share data, the operating system and library functions must ensure that only one thread is reading or writing data at one time. The multithread libraries provide this support. (You can use these libraries for single-thread programs, but the multithread support causes unnecessary overhead.)

To indicate that you want the multithread libraries, specify the `/Gm+` compiler option. For example:

```
icc /Gm+ mymulti.c
```

Conversely, the `/Gm-` option, which is the default, specifies explicitly to use the single-thread version of the library.

If you intend to compile your source code into separate modules and then link them into one executable program file, you must compile each module using the `/Gm+` option and ensure that the multithread libraries are used when you link them. You cannot mix modules that have been compiled with `/Gm+` with modules compiled using `/Gm-`.

You can use either static (`/Gd-`) or dynamic (`/Gd+`) linking with multithread programs.

Sample Multithread Program

If you installed the sample programs, there will be created in your VisualAge C++ `Samples` folder, a project called `SAMPLE2A` and a second project called `SAMPLE2B`. Make and debug these projects to see how they work. (For information on making and debugging projects, see the *User's Guide*.)

Alternatively, you can build and run the samples from the command line. The files for `SAMPLE2A` and `SAMPLE2B` are found in the `\IBMCPP\SAMPLES\COMPILER\SAMPLE02` directory under the main VisualAge C++ directory. The `read.me` file in that directory describes the process for running the programs from the command line.

Sample Multithread Program

Building Dynamic Link Libraries

Dynamic linking is the process of resolving references to external data and code at runtime or loadtime instead of at link time. A dynamic link library is an object module which can be shared by more than one process. You can dynamically link with the supplied VisualAge C++ runtime DLLs, as well as with your own DLLs. The advantages of using a dynamic link library include:

- smaller memory requirement as several applications can all share the same dynamic link library instead of each application having its own copy of the functions contained in the DLL.
- simplified application modification because modifications to an application's object module does not necessitate recompilation of the DLL.
- flexible software support as DLL object modules can be replaced with newly released, improved versions without forcing recompilation of the application code.

There are basically two types of dynamic link libraries (DLLs) -- those which contain code and those which do not. An example of the latter are resource DLL's which contain no code, only resources such as menus or icons that are used by Presentation Manager. Dynamic Link Libraries which contain code can be further classified along three dimensions: whether they statically or dynamically link to the VisualAge C++ runtime, whether they support multithread or only singlethread executables, and whether they use the full system or only the subsystem libraries.

This chapter describes the steps for creating and using a dynamic link library:

1. Creating the source files for a DLL
2. Creating a module definition file (`.DEF`) for the DLL
3. Compiling the source files and linking the resulting object files to build a `.DLL` file
4. Using the DLL. When an application which uses the DLL is linked, the linker must be informed that there are references to functions and/or variables that will not be resolved until runtime. The linker is informed either by indicating an import library file (`.LIB`) for the DLL, or by indicating a module definition file for the external module to be used when linking. The section of this chapter on using your DLL, explains how you create the library file and the the module definition file.

This chapter also provides information on: how to create your own DLL initialization and termination function, your own library DLLs, and your own resource DLLs.

Examples are provided throughout the chapter to illustrate the process.

The examples shown are from the `SAMPLE03` project. If you installed the sample programs, you will find the `SAMPLE03` project in the VisualAge C++ `Samples` folder. For information on how to build and debug a project, see the *User's Guide* Alternatively, you can compile, link, and run the sample from the command line. The files for `SAMPLE03` are found in the `\IBMCPP\SAMPLES\COMPILER\SAMPLE03` directory under the main VisualAge C++ directory, along with a `readme` file that describes the process.

Creating DLL Source Files

To build a DLL, you must first create source files containing the data and/or functions that you want to include in your DLL. No special file extension is required for DLL source files. The source code can be written in C or C++.

Each function that you want to *export* from the DLL (that is, a function that you plan to call from other executable modules or DLLs) must be an external function, either by default or by being qualified with the `extern` keyword. Otherwise, the linker will not find your function references and will generate errors.

If your DLL and the modules that access it do not dynamically link to the same runtime DLL, you must use the **`#pragma handler`** directive to ensure OS/2 exceptions are handled properly within your DLL. Use **`#pragma handler`** at the entry point of each DLL function to register the library exception handler `_Exception`. On exit from the function, code will also be generated to deregister `_Exception`.

Note: You need to explicitly register the exception handler only for the functions that will be exported from the DLL.

For more information on **`#pragma handler`**, see the online *Language Reference*. For information on exception handling, see Chapter 14, “Signal and OS/2 Exception Handling” on page 217.

Example of a DLL Source File

The file `SAMPLE03.C` is the source file for the DLL used in the `SAMPLE03` project in the VisualAge C++ `SAMPLES` folder. The file can be found in the `\IBMCPP\SAMPLES\COMPILER\SAMPLE03` directory under the main VisualAge C++ directory.

The source file contains the code for:

- Three sorting functions: `bubble`, `insertion`, and `selection`
- Two static functions, `swap` and `compare`, that are called by the sorting functions
- A function, `list`, that lists the contents of an array.

Creating a Module Definition File

A module definition (`.DEF`) file is a plain text file that describes the names, attributes, exports, imports, and other characteristics of an application or dynamic link library. You must use a module definition file when you create any OS/2 DLL.

Example of a Module Definition File

The `.DEF` file for the `SAMPLE03` program is shown here to illustrate the most common statements used in a module definition file to build DLLs. For a complete description of module definition files, refer to the *User's Guide* for the VisualAge C++ linker utility.

```
LIBRARY SAMPLE03 INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE NONSHARED READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
   nSize     ; array size
   pArray    ; pointer to base of array of ints
   nSwaps    ; number of swaps required to sort the array
   nCompares ; number of comparisons required to sort the array
   list      ; array listing function
   bubble    ; bubble sort function
   insertion ; insertion sort function
```

Figure 5. SAMPLE03.DEF - DLL Module Definition File

Note: In this module definition file, the `EXPORTS` statement does not include the `selection` function because the source code contains a **`#pragma export`** statement for `selection`.

The module statements specified in the .DEF file are as follows:

LIBRARY SAMPLE03 INITINSTANCE TERMINSTANCE

This statement identifies the executable file as a dynamic link library and specifies that SAMPLE03 is the name of the DLL. It also uses the following attributes to specify when the _DLL_InitTerm function will be called:

INITINSTANCE

The function is called the first time the DLL is loaded for each process that accesses the DLL. The alternative is INITGLOBAL; the function is called only the first time the DLL is loaded.

INITGLOBAL is the default.

TERMINSTANCE

The function is called the last time the DLL is freed for each process that accesses the DLL. The alternative is TERMGLOBAL; the function is called only the final time the DLL is freed.

TERMGLOBAL is the default.

PROTMODE

This statement specifies that the DLL can be run in protected (OS/2) mode only.

DATA MULTIPLE NONSHARED READWRITE LOADONCALL

This statement defines the default attributes for data segments within the DLL. The attributes are:

MULTIPLE

Specifies that there is a unique copy of the data segment for each process. The alternative is SINGLE; there is only one data segment for all processes to share.

SINGLE is the default for DLL's.

MULTIPLE is the default for applications.

NONSHARED

Specifies that the data segment cannot be shared and must be loaded separately for each process. The alternative is SHARED; one copy of the segment is loaded and shared by all processes that access the module.

SHARED is the default for DLLs, NONSHARED is the default for applications.

Note that if you use the READONLY attribute, data segments are always shared.

Note: If the two above data segment attributes conflict, such as `SINGLE NONSHARED` or `MULTIPLE SHARED`, the behaviour is undefined.

`READWRITE`
: Means that you can read from or write to the data segment. The alternative is `READONLY`; you can only read from the data segment.

`READWRITE` is the default.

`LOADONCALL`
: Means that the data segment is loaded into memory when it is first accessed. `LOADONCALL` is the default.

Note: You can also specify `PRELOAD`, but Version 2.0 and later of the OS/2 operating system ignore the `PRELOAD` attribute and use the `LOADONCALL` instead.

See "Defining Code and Data Segments" on page 66 for information on defining your own data segments.

`CODE LOADONCALL`
: This statement defines the default attributes for code segments within the DLL. `LOADONCALL` means that the code segment is loaded when it is first accessed. `LOADONCALL` is the default.

Note: You can also specify `PRELOAD`, but Version 2.0 and later of the OS/2 operating system ignore the `PRELOAD` attribute and use the `LOADONCALL` instead.

For information on defining your own code segments, see "Defining Code and Data Segments" on page 66.

`EXPORTS`
: This statement defines the names of the functions and variables to be exported to other runtime modules. Following the `EXPORTS` keyword are the export definitions, which are simply the names of the functions and variables that you want to export. Each name must be entered on a separate line. See "Defining Functions to be Exported" on page 66 for more information.

Note: When you build your DLL using `/Gd-`, so that it is statically linked to the runtime library, you must specify the following attributes in your `.DEF` file:

INITINSTANCE
TERMINSTANCE
DATA MULTIPLE NONSHARED

Defining Code and Data Segments

In the .DEF file shown, all data and code segments are given the same attributes. If you want to specify different attributes for different sets of data or code, you can use the **#pragma data_seg** and **#pragma alloc_text** directives to define your own data and code segments, or the /Nd and /Nt compiler options to specify the name of the default data or code segments, respectively. You can then list the segments in the .DEF file under the heading SEGMENTS, and specify attributes for each. For example:

```
SEGMENTS
   mydata SHARED READONLY
   mycode PRELOAD
```

Any segments that you do not specify under SEGMENTS are given the attributes specified by the DATA or CODE statement, depending on the type of segment.

For more information about **#pragma data_seg** and **pragma alloc_text**, see the online *Language Reference*. The /Nd and /Nt options are described in the *User's Guide*.

Defining Functions to be Exported

When you export a function from a DLL, you make it available to programs that call the DLL. If you do not export a function, it can only be used within the DLL itself.

To export a function, list its name under the EXPORTS keyword in the .DEF file as described below. Note that if your DLL is written in C++, you must specify the *mangled* or encoded name of the function. A utility is provided to assist you with this task. For an explanation of how to use the CPPFILT utility to mangle and demangle function names, see "Demangling (Decoding) C++ Function Names" on page 394.

You can also use **#pragma export** or the **_Export** keyword to specify that a function is to be exported. For example, in SAMPLE03.C, the function selection is declared to be exported by a **#pragma export** directive. The **#pragma** directive also allows you to specify the name the exported function will have outside of the DLL and its ordinal number. When you use the keyword or **#pragma** directive for C++ functions, use the normal function name, not the encoded name.

If you use **`#pragma export`** or **_Export** to export your function, you may still need to provide an `EXPORTS` entry for that function. If your function has all of the following default characteristics

- Has no I/O privileges, and
- Is exported by ordinal. (If this is the case, you do not want the system loader to also keep its name resident in memory.)

it does not require an `EXPORTS` entry. If your function has characteristics other than the defaults, the only way you can specify them is with an `EXPORTS` entry in your `.DEF` file.

For more information about **_Export** and **`#pragma export`**, see the online *Language Reference*.

Additional C++ Considerations

For C++ DLLs, ensure that you export all member functions that are required. If an inlined or exported function uses private or protected members, you must also export those members. In addition, you should export all static data members. If you do not export the static data members of a particular class, users of that class cannot debug their code because the reference to the static data members cannot be resolved.

Compiling and Linking Your DLL

To compile your source files to create a DLL, use the `/Ge-` compiler option. You may also want to use the `/C+` option to compile your files without linking them, and then link them separately.

You must also specify the runtime libraries you want to use:

- Single-thread (`/Gm-`) or multithread (`/Gm+`). See Chapter 5, "Creating Multithread Programs" on page 47 for information on multithread libraries.
- Statically linked (`/Gd-`) or dynamically linked (`/Gd+`). See the *User's Guide* for more information on static and dynamic linking.

  Note: The method of linking used for the runtime libraries is independent of the module type you create; you can statically link the runtime functions in a dynamic link library.

For more information on compiler options, see the *User's Guide*.

If your DLL contains C++ code that uses templates, there are additional considerations. See "Creating C++ DLLs" on page 69 for details on creating a C++ DLL.

Compiling and Linking Your DLL

When you use `icc` to compile and link your DLL, you must specify on the command line all the DLL source files followed by the module definition file. The name of the first source file (without the file name extension) is used as the name of the DLL.

For example, to compile and link the files `mydlla.c` and `mydllb.c`, using the `mydll.def` module definition file, use the command:

```
icc /Ge- mydlla.c mydllb.c mydll.def
```

The resulting DLL will be called `mydlla.dll`.

Note: The `/Ge-` option tells the compiler you are building a DLL, rather than an executable file. The options to indicate the single-thread library (`/Gm-`) and to link the runtime libraries statically (`/Gd-`) are the defaults.

If you are compiling and linking separately, you must give the following information to the VisualAge C++ linker:

- The compiled object (`.OBJ`) files for the DLL
- The name to give the DLL
- The C libraries to use
- The name of the module definition file.

Note: The compiler includes information in the object files on the C libraries you indicated by the compiler options that control code generation (see the *User's Guide*). These libraries are automatically used at link time. You do not need to specify C runtime libraries on the linker command line unless you want to override the ones you chose at compile time.

For example, the following commands: create the DLL `finaldll.dll`.

```
icc /C+ /Ge- mydlla.c mydllb.c
  ILINK /NOI mydlla.obj mydllb.obj mydll.def /OUT:finaldll.dll
```

These commands:

- Compile the source files `mydlla.c` and `mydllb.c`
- Link the resulting object files with the single-thread, statically linked C libraries, using the definition file `mydll.def`

The preferred method is to use `icc` to both compile and invoke the linker for you You could use `icc` to both compile and invoke the linker for you with the following command:

```
icc /Ge- /Fefinal.dll mydlla.c mydllb.c mydll.def
```

Note: The `icc` command passes the linker option `/NOI` to the linker by default. The `/NOI` option preserves the case of external names.

Creating C++ DLLs

When your DLL is written in C++, there are considerations that do not apply to DLLs written in C. You must ensure that classes and class members are exported correctly, especially if they use templates.

You can build C++ DLLs in one of two ways:

- Using the **_Export** keyword and **#pragma export**
- Using the CPPFILT utility to create a .DEF file.

The SAMPLE07 project provides examples of both methods of building C++ DLLs. The files for SAMPLE07 can be found in the \IBMCPP\SAMPLES\COMPILER\SAMPLE07 directory.

Using _Export and #pragma export

This is the simplest method of creating a C++ DLL:

1. Use **_Export** or **#pragma export** in your source files to specify the classes and functions (including member functions) that you want to export from your DLL. For example:

   ```
   class triangle : public area
   {
      public:
         static int _Export objectCount;
         double _Export getarea();
         _Export triangle::triangle(void);
   };
   ```

   exports the getarea function and the constructor for class triangle. Alternatively, you could use **#pragma export**:

   ```
   #pragma export(triangle::objectCount(),,1)
   #pragma export(triangle::getarea(),,1)
   #pragma export(triangle::triangle(void),,2)
   ```

   Important: You must always export constructors and destructors.

   The **_Export** keyword and **#pragma** directive are described in more detail in the online *Language Reference*.

2. Create a .DEF file as described in "Creating a Module Definition File" on page 63. Do not specify any entries under EXPORTS.

3. Use icc to compile and link the DLL. If you use any of the Complex, Collection, or User Interface class libraries, you must specify the library names on the command line for the link step. If you link in a separate step, you must also specify the /Tdp option.

C++ DLLs

The SAMPLE07, METHOD1 project in the VisualAge C++ SAMPLES folder demonstrates this method. Build and run it to see how it works. For directions on building and running a project, see the *User's Guide*.

Alternatively, you can compile and link this sample from the command line. The SAMPLE07 files corresponding to the SAMPLE07, METHOD1 project can be found in the \IBMCPP\SAMPLES\COMPILER\SAMPLE07 directory under the main VisualAge C++ directory. The readme file in that directory gives instructions for running the sample from the command line.

Using CPPFILT

To build a DLL using the CPPFILT utility:

1. Compile your source files as you would for any DLL.
2. If you use templates, compile the template-include files located in the TEMPINC directory under the source directory. These files contain the implementation of all instantiated templates that are used in the files you compiled and are needed when you link your DLL.
3. Copy the objects created from the template-include files into the directory with your other DLL objects.
4. Run CPPFILT on all your object files together. Because CPPFILT sends output to **stdout**, ensure you redirect the output to a file. For example:

   ```
   CPPFILT /B /P file1.obj file2.obj > cppdll.def
   ```

   The /B option specifies that the files are binary, and the /P option specifies to include all public symbols in the CPPFILT output. For more details on the CPPFILT utility, see "Using the CPPFILT Utility" on page 395.
5. Edit the output file. Delete entries for functions and variables that you do not want to export from your DLL. Then create a .DEF file, specifying the remaining entries under the EXPORTS heading.
6. Use icc to link your objects, libraries, and .DEF file into a DLL. If you use any of the Complex, Collection, or User Interface classes, you must specify the library names on the command line. You must also specify the /Tdp option.
7. Erase the template-include objects that you have included in the DLL so they are not linked into any applications that use your DLL. Alternatively, use the /Ft- option when you link the accessing applications. If these objects are included more than once, the linker will generate error messages about multiply-defined symbols.

The SAMPLE07, METHOD2 project in the VisualAge C++ SAMPLES folder demonstrates this method. Build and run the project to see how it works. For directions on building and running a project, see the *User's Guide*.

Alternatively, you can compile and link this sample from the command line. To compile and link this sample, follow the directions contained in the readme file in the \IBMCPP\SAMPLES\COMPILER\SAMPLE07 directory. The SAMPLE07 files corresponding to the SAMPLE07, METHOD2 project can be found in the \IBMCPP\SAMPLES\COMPILER\SAMPLE07 directory under the main VisualAge C++ directory.

Exporting Virtual Function Tables from a DLL

Follow these steps to export a VFT from a DLL:

- A virtual function table (VFT) is usually generated in the compilation unit that defines the first non-inline virtual function in a class. You can use the /Wvft option to find out which function that is. The object file that contains the definition for this function will also contain the VFT.
- Once you know which object file contains the VFT, you can use CPPFILT to dump the symbols in the object file. One of these symbols will be the name of the VFT that you want to export.
- After you have determined what the name of the VFT is, you can either use the output of CPPFILT directly in the .DEF file or you can manually add an entry for the VFT in the .DEF file.

An example of the symbols dumped by CPPFILT

```
;From object file:  os2prod\object.obj
  ;PUBDEFs (Symbols available from object file):
    ;  ComentObjectRecord::OMFExtensions::getRecordData() const
    getRecordData__Q2_18ComentObjectRecord13OMFExtensionsCFv
    ;  ObjRecNameField::ObjRecNameField(const char*)
    __ct__15ObjRecNameFieldFPCc
    ;  PubDef16ObjectRecord::getRecordData(unsigned char*&) const
    getRecordData__20PubDef16ObjectRecordCFRPUc
    ;  LNamesObjectRecord::LNamesObjectRecord(unsigned int,const unsigned char*)
    __ct__20PubDef16ObjectRecordFUlPCUc
    ;  {PubDef32ObjectRecord}ObjectRecord::virtual-fn-table-ptr
    __vft20PubDef32ObjectRecord12ObjectRecord
    ;  PubDef32ObjectRecord::PubDef32ObjectRecord(unsigned long,const unsigned char*)
    setNameString__15ObjRecNameFieldFPCc
    ;  PubDef32ObjectRecord::getRecordData(unsigned char*&) const
    getRecordData__20PubDef32ObjectRecordCFRPUc
    ;  {ComDefObjectRecord}ObjectRecord::virtual-fn-table-ptr
    __vft18ComDefObjectRecord12ObjectRecord
```

An example of VFTs in the .DEF file.

```
NAME object   WINDOWCOMPAT
PROTMODE

IMPORTS
 ;  {PubDef32ObjectRecord}ObjectRecord::virtual-fn-table-ptr
 __vft20PubDef32ObjectRecord12ObjectRecord
 ;  {ComDefObjectRecord}ObjectRecord::virtual-fn-table-ptr
 __vft18ComDefObjectRecord12ObjectRecord
```

Using Your DLL

Write the source files that are to access your DLL as if the functions and/or variables are to be statically linked at compile time. Then when you link the program, you must inform the linker that some function and/or variable references are to a DLL and will be resolved at run time. There are two ways to communicate this information to the linker:

1. Use the IMPLIB utility to create a library file with all the information that the linker needs about the DLL. The IMPLIB utility uses a module definition file to create an import library (`.LIB`) file for the DLL. When you link an executable module, the linker uses this import library to resolve external references to the DLL.

   If your DLL contains any C++ templates, you must always access the DLL by means of an import library to ensure that the names you use when you instantiate the template are resolved correctly.

   If you invoke the linker directly, give the name of the import library where you normally specify library names. For example:

   ```
      ILINK /NOI mymain.obj finaldll.lib;
   ```

   If you invoke the linker through the `icc` command, you must put the name of the import library in the compiler invocation string. For example:

   ```
   icc mymain.c finaldll.lib
   ```

   See the *User's Guide* for more information on IMPLIB.

   Note: The import libraries for the VisualAge C++ runtime DLLs have been supplied with the compiler.

2. Construct a module definition file for the accessing module that is being linked. The definition file specifies which variables and names will be obtained from a DLL at run time, and in which DLLs these items will be found. In general, import libraries are easier to use and maintain than module definition files.

Note: To make functions in a DLL available to other programs, the name of those functions must have been exported (using **`#pragma export`** or the **_Export** keyword in the source file, or with an `EXPORT` entry in the `.DEF` file) when the DLL was linked. Also, all DLLs must be in a directory listed in the `LIBPATH` environment variable (as described in Chapter 1, "Setting Runtime Environment Variables" on page 3).

Deciding the Best Way to Export Functions from Your DLL

There are three different ways to export the functions in a DLL so that they are available to other programs:

1. Using a `.DEF` file. With this method, the functions are exported by name. It can be difficult to write and maintain C++ `.DEF` files because you must use the mangled names of the functions that you want to export.
2. Using the **_Export** keyword in the source files. With this method, you can only export the functions by ordinal. Furthermore, you cannot control which ordinal is assigned to a particular function. This is the easiest method for exporting functions, but it can cause problems if other programs that use the DLL depend on a particular set of ordinals. If the DLL has to be updated, the compiler may assign different ordinals to the exported functions.
3. Using **`#pragma export`** in the source files. With this method, you can only export the functions by ordinal, but you have the choice of choosing the ordinal for a function yourself or letting the compiler choose it for you. With this method, you can specify the assignment of ordinals to exported functions. When you update a DLL, you can keep these assignments. This means that programs that use functions from this DLL will not have to be updated when the DLL is updated.

The advantage of using a `.DEF` file to export functions is that changing the DLL will not affect other programs that use functions in the DLL. The disadvantage of using a `.DEF` file is that the load time can be greater for the code that uses the DLL, and it takes time to create the `.DEF` file itself.

Sample Definition File for an Executable Module

The following figure shows the module definition file used for the `main` program in the sample project `SAMPLE03`.

```
NAME MAIN03 WINDOWCOMPAT

IMPORTS
   SAMPLE03.nSize
   SAMPLE03.pArray
   SAMPLE03.nSwaps
   SAMPLE03.nCompares
   SAMPLE03.list
   SAMPLE03.bubble
   SAMPLE03.insertion
```

Figure 6. MAIN03.DEF - Definition File for an Executable Module

Note: There is no statement to import the `selection` function because it is imported using **`#pragma import`** statement in the source code.

The statements given are as follows:

`NAME MAIN03 WINDOWCOMPAT`
: The `NAME` statement assigns the name `MAIN03` to the program being defined. If no name is given, the name of the executable module (without the `.EXE` extension) is used. `WINDOWCOMPAT` specifies that the program is compatible with the PM environment. The alternatives are `NOTWINDOWCOMPAT`, which means the program is not compatible with the PM environment, or `WINDOWAPI`, which means the program uses PM APIs.

`IMPORTS`
: This statement defines the names of functions and variables to be imported for the program. Following the `IMPORTS` keyword are the import definitions. Each definition consists of the name of the DLL where the function or variable is found and the name of the function or variable. The two names must be separated by a period, and each definition must be entered on a separate line.

You can also use **#pragma import** to specify that a function is imported from a DLL. You can use this **#pragma** directive to import the function by name or by ordinal number.

For a detailed description of **#pragma import**, see the online *Language Reference*. For an example of using this **pragma**, see MAIN03.C, the main program for the SAMPLE03 project.

Initializing and Terminating the DLL Environment

The initialization and termination entry point for a DLL is the _DLL_InitTerm function. When each new process gains access to the DLL, this function initializes the necessary environment for the DLL, including storage, semaphores, and variables. When each process frees its access to the DLL, the _DLL_InitTerm function terminates the DLL environment created for that process.

The default _DLL_InitTerm function supplied by VisualAge C++ compiler performs the actions required to initialize and terminate the runtime environment. It is called automatically when you link to the DLL.

If you require additional initialization or termination actions for your runtime environment, you will need to write your own _DLL_InitTerm function. For more information, see "Writing Your Own _DLL_InitTerm Function" on page 77. A sample _DLL_InitTerm function is included for the SAMPLE03 project (see "Example of a User-Created _DLL_InitTerm Function" on page 79.)

Note: The _DLL_InitTerm function provided in the subsystem library differs from the runtime version. See "Building a Subsystem DLL" on page 208 for more information about building subsystem DLLs.

Sample Program to Build a DLL

The sample project SAMPLE03 shows how to build and use a DLL that contains three different sorting functions. These functions keep track of the number of swap and compare operations required to do the sorting.

The files for the sample program are:

SAMPLE03.C	The source file for the DLL, described in “Example of a DLL Source File” on page 62.
INITTERM.C	The _DLL_InitTerm function, shown in “Example of a User-Created _DLL_InitTerm Function” on page 79.
SAMPLE03.DEF	The module definition file for the DLL, shown in “Creating a Module Definition File” on page 63.
MAIN03.DEF	The module definition file for the executable, shown in “Sample Definition File for an Executable Module” on page 74.
SAMPLE03.H	The user include file.
MAIN03.C	The main program.

If you installed the sample programs, these files are found in the \IBMCPP\SAMPLES\COMPILER\SAMPLE03 directory under the main VisualAge C++ directory. Two make files that build the sample are also provided, MAKE03S for static linking and MAKE03D for dynamic linking.

Note: You must have the Toolkit installed to use the make files.

To compile and link this sample program, from the \IBMCPP\SAMPLES\COMPILER\SAMPLE03 directory, use NMAKE with the appropriate make file. For example:

```
nmake all /f MAKE03S
```

To compile and link the program yourself, use the following commands:

Command	**Description**
`icc /Ge- /B"/NOE" /DSTATIC_LINK SAMPLE03.C INITTERM.C SAMPLE03.DEF`	Compiles and links `SAMPLE03.C` using default options and • Creates a DLL (`/Ge-`) • Passes the `/NOE` option to the linker (see description below) • Defines `STATIC_LINK`.
`icc MAIN03.C MAIN03.DEF`	Compiles `MAIN03.C` using default options.

The `/NOE` linker option tells the linker to ignore the extended library information found in the object files. The linker then uses the version of `_DLL_InitTerm` that you provide instead of the one from the VisualAge C++ runtime library.

To run the program, enter `MAIN03`.

Writing Your Own _DLL_InitTerm Function

If your DLL requires initialization or termination actions in addition to the actions performed for the runtime environment, you will need to create your own `_DLL_InitTerm` function. The prototype for the `_DLL_InitTerm` function is:

```
unsigned long _System. _DLL_InitTerm(unsigned long modhandle,
                                     unsigned long flag);
```

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended.

The *modhandle* parameter is the module handle assigned by the operating system for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, `DosQueryModuleName` can be used to return the fully qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from `_DLL_InitTerm` tells the loader if the initialization or termination 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.

Because it is called by the operating system loader, the `_DLL_InitTerm` function must be compiled using **_System** linkage.

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 an example of a _DLL_InitTerm function for a subsystem DLL, see "Example of a Subsystem _DLL_InitTerm Function" on page 209.

Initializing the Environment

Before you can call any VisualAge C++ library functions, you must first initialize the runtime environment. Use the function _CRT_init, which is provided in the runtime libraries. The prototype for this function is:

```
int _CRT_init(void);
```

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**.

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(void);
```

Note: If you are providing your own version of the _matherr function to be used in your DLL, you must 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. The prototype for this function is:

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

The parameter required is the address of your _matherr function.

Terminating the Environment

If your DLL is statically linked, you must use the _CRT_term function to correctly terminate the C runtime environment. The _CRT_term function is provided in the VisualAge C++ runtime libraries. It has the following prototype:

```
void _CRT_term(void);
```

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(void);
```

Once you have called _CRT_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. If your termination routine requires calling library functions, you must register the termination routine with DosExitList. Note that all DosExitList routines are called before DLL termination routines.

Example of a User-Created _DLL_InitTerm Function

The following figure shows the _DLL_InitTerm function for the sample project SAMPLE03.

```
#define  INCL_DOSMODULEMGR
#define  INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* _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

/* A clean up routine registered with DosExitList must be used if runtime   */
/* calls are required and the runtime is dynamically linked.  This will     */
/* guarantee that this clean up routine is run before the library DLL is    */
/* terminated.                                                              */

static void _System cleanup(ULONG ulReason);
#endif
size_t nSize;
int *pArray;
```

Figure 7 (Part 1 of 3). INITTERM.C - _DLL_InitTerm *Function for* SAMPLE03

```
/* _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)
{
   size_t i;
   APIRET rc;
   char namebuf[CCHMAXPATH];

   /* If ulFlag is zero then the DLL is being loaded so initialization should*/
   /* be performed.  If ulFlag is 1 then the DLL is being freed so          */
   /* termination should be performed.                                      */

   switch (ulFlag) {
      case 0 :

         /*******************************************************************/
         /* 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

         /*******************************************************************/
         /* A DosExitList routine must be used to clean up if runtime calls */
         /* are required and the runtime is dynamically linked.             */
         /*******************************************************************/

            if (rc = DosExitList(0x0000FF00|EXLST_ADD, cleanup))
            printf("DosExitList returned %lu\n", rc);
#endif
         if (rc = DosQueryModuleName(hModule, CCHMAXPATH, namebuf))
            printf("DosQueryModuleName returned %lu\n", rc);
         else
            printf("The name of this DLL is %s\n", namebuf);
```

Figure 7 (Part 2 of 3). INITTERM.C - _DLL_InitTerm *Function for* SAMPLE03

```
         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)
            pArray[i] = rand();
         break;
      case 1 :
#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(ULONG ulReason)
{
   if (!ulReason) {
      printf("The array will now be freed.\n");
      free(pArray);
   }
   DosExitList(EXLST_EXIT, cleanup);
   return ;
}
#endif
```

Figure 7 (Part 3 of 3). INITTERM.C - `_DLL_InitTerm` *Function for* `SAMPLE03`

The SAMPLE03 program is described in more detail in “Sample Program to Build a DLL” on page 76.

Creating Resource DLLs

Resource DLLs contain application resources that your program uses, such as menus, bitmaps, and dialog templates. You can define these resources in a `.RC` file using OS/2 APIs, or with the Icon Editor and Dialog Editor. Use the Resource Compiler to build the resources into a DLL, which is then called by your executable program at run time.

One of the benefits of using a resource DLL instead of binding the resources directly into your executable file includes easier maintenance and less duplication of resources. You may even be able to use a common resource DLL for multiple applications.

Another benefit is that you can completely isolate your program from your resources. Translations can be made to your resource DLL without your program having to be recompiled or linked, or even re-bound to run in the new language. Alternatively, you can create a different resource DLL per locale or codepage setting and load either the most appropriate one based on your locale and codepage setting, or load the default one if a specific one is not available.

For instance, you could have three resource DLLs: my844.DLL, my029.DLL, and myDef.dll. If running in codepage 844, load my844.DLL; in codepage 029, load my029.DLL; else load mydef.DLL. Again, no changes are required to the program to work in a potentially endless number of codepages.

To create a resource DLL:

1. Create an empty source file. By "empty" we mean a file with no code, no declarations or the like. The file must be empty because it is being included in a resource DLL and a resource DLL can contain only resources.
2. Create a `.DEF` file. The only statement required in this file is `LIBRARY` to specify that a DLL is to be built.
3. Create a `.RC` file that defines your resources.
4. Compile the source file using `/C+` to specify compile only. For example:

   ```
   icc /C+ empty.c
   ```

   Do not specify the `/Ge-` option. Specifying `/Ge-` causes the DLL initialization and termination code to be included in the object module, and the resource DLL cannot contain code.
5. Link the resulting object module, using your `.DEF` file, to create an empty DLL:

   ```
    ILINK empty.obj mydef.def /OUT:resdll.dll
   ```

6. Compile your `.RC` file with the Resource Compiler to create a `.RES` file. For example:

   ```
   RC /r myres.rc
   ```

7. Use the Resource Compiler again to add the resources to the DLL. For example:

   ```
   RC myres.res resdll.dll
   ```

Your application can use OS/2 APIs to load the resource DLL and access the resources it contains. Like other DLLs, resource DLLs must be in a directory specified in your LIBPATH environment variable.

📖 For more information on resources and the Resource Compiler, see the *User's Guide* and *Tools Reference*.

Creating Your Own Runtime Library DLLs

If you are shipping your application to other users, you must use one of three methods to make the VisualAge C++ runtime library functions available to the users of your application:

1. Statically bind every module to the library (`.LIB`) files.

   This method increases the size of your modules and also slows the performance because the library environment has to be initialized for each module. Having multiple library environments also makes signal handling, file I/O, and other operations more complicated.

2. Use the DLLRNAME utility to rename the VisualAge C++ library DLLs.

   You can then ship the renamed DLLs with your application. 📖 DLLRNAME is described in the *User's Guide*.

3. Create your own runtime DLLs.

   This method provides one common runtime environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the VisualAge C++ DLLs change, you need only rebuild your own DLL. In addition, you can tailor your runtime DLL to contain only those functions you use, including your own.

Creating Runtime Library DLLs

To create your own runtime library, follow these steps:

1. Copy and rename the appropriate VisualAge C++ `.DEF` file for the program you are creating. For example, for a multithread program, copy `CPPOM30.DEF` to `myrtdll.def`. You must also change the DLL name on the `LIBRARY` line of the `.DEF` file. The `.DEF` files are installed in the LIB subdirectory under the main VisualAge C++ installation directory.
2. Remove any functions that you do not use directly or indirectly (through other functions) from your `.DEF` file (`myrtdll.def`), file, including the `STUB` line. Do not delete anything with the comment `****` next to it; variables and functions indicated by this comment are used by startup functions and are always required.
3. Create a source file for your DLL, for example, `myrtdll.c`. If you are creating a runtime library that contains only VisualAge C++ functions, create an empty source file. If you are adding your own functions to the library, put the code for them in this file.
4. Compile and link your DLL files. Use the `/Ge-` option to create a DLL, and the appropriate option for the type of DLL you are building (single-thread or multithread). For example, to create a multithread DLL, use the command:

   ```
   icc /Ge- /Gm+ myrtdll.c myrtdll.def
   ```
5. Use the IMPLIB utility to create an import library for your DLL, as described in "Using Your DLL" on page 72. For example:

   ```
   IMPLIB /NOI myrtdlli.lib myrtdll.def
   ```
6. Use the ILIB utility to add the object modules that contain the initialization and termination functions to your import library. These objects are needed by all executable modules and DLLs. They are contained in `CPPOM300.LIB` for multithread programs and `CPPOS300.LIB` for single-thread programs.

   See the *User's Guide* online documentation for information on how to use ILIB.

   Note: If you do not use the ILIB utility, you must ensure that all objects that access your runtime DLL are statically linked to the appropriate object library.

7. Compile your executable modules and other DLLs with the `/Gn+` option to exclude the default library information. For example:

   ```
   icc /C /Gn+ /Ge+ myprog.c
   icc /C /Gn+ /Ge- mydll.c
   ```

   When you link your objects, specify your own import library. If you are using or plan to use OS/2 APIs, specify OS2386.LIB also. For example:

   ```
     ILINK myprog.obj myrtdlli.lib OS2386.LIB
     ILINK mydll.obj myrtdlli.lib OS2386.LIB
   ```

   To compile and link in one step, use the commands:

   ```
   icc /Gn+ /Ge+ myprog.c myrtdlli.lib OS2386.LIB
   icc /Gn+ /Ge- mydll.c myrtdlli.lib OS2386.LIB
   ```

   Note: If you did not use the ILIB utility to add the initialization and termination objects to your import library, specify the following when you link your modules:

   a. `CPPOS300.LIB` or `CPPOM300.LIB`
   b. Your import library
   c. OS2386.LIB (to allow you to use OS/2 APIs)
   d. The linker option `/NOD`.

   For example:

   ```
   ILINK /NOD myprog.obj CPPOS300.LIB myrtdlli.lib OS2386.LIB;
   ILINK /NOD mydll.obj CPPOS300.LIB myrtdlli.lib OS2386.LIB;
   ```

   The `/NOD` option tells the linker to disregard the default libraries specified in the object files and use only the libraries given on the command line. If you are using `icc` to invoke the linker for you, the commands would be:

   ```
   icc /B"/NOD" myprog.c CPPOS300.LIB myrtdlli.lib OS2386.LIB
   icc /Ge- /B"/NOD" mydll.c CPPOS300.LIB myrtdlli.lib OS2386.LIB
   ```

   The linker then links the objects from the object library directly into your executable module or DLL.

Example of Creating a Runtime Library

In the sample project SAMPLE03, the program MAIN03.C calls printf and srand from the VisualAge C++ runtime DLLs, and uses other variables and functions from SAMPLE03.DLL. Because SAMPLE03.DLL also uses printf and is statically linked to the runtime libraries, the code for the VisualAge C++ runtime functions it uses is linked into SAMPLE03.DLL.

If these functions are included in SAMPLE03.DLL, all external references from MAIN03.C can be resolved by dynamically linking to this DLL. As a result, MAIN03.EXE will be smaller.

Note: The process described here is only possible when the user DLL links statically to the VisualAge C++ runtime library.

Rebuild SAMPLE03.DLL to include printf and srand as exports by following these steps:

1. Add _printfieee and srand to SAMPLE03.DEF under the EXPORTS keyword.

   Note: When the language level is /Se, printf is mapped to _printfieee to support the IEEE extensions (infinity and NaN).

2. Use CPPOS30.DEF to find what functions and variables must be exported, and add them to SAMPLE03.DEF as EXPORTS.
3. Relink SAMPLE03.DLL as described in "Compiling and Linking Your DLL" on page 67.

After your changes, SAMPLE03.DEF should look like Figure 8. The example shown in this figure is actually the file SAMPLE3R.DEF, which is provided with the SAMPLE03 project.

```
LIBRARY SAMPLE03 INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE NONSHARED READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
   nSize     ; array size
   pArray    ; pointer to base of array of ints
   nSwaps    ; number of swaps required to sort the array
```

Figure 8 (Part 1 of 2). SAMPLE3R.DEF - Definition File to Export C Runtime Functions

```
nCompares ; number of comparisons required to sort the array
list      ; array listing function
bubble    ; bubble sort function
selection ; selection sort function
insertion ; insertion sort function
          ; CRT symbols required by EXE
_printfieee
srand
_critlib_except                           ; ****
_DosSelToFlat                             ; ****
_DosFlatToSel                             ; ****
_environ                                  ; ****
_CRT_init                                 ; ****
__ctordtorInit                            ; ****
_EXE_Exception                            ; ****
_Exception                                ; ****
_PrintErrMsg                              ; ****
_exception_procinit                       ; ****
_exception_dllinit                        ; ****
_matherr                                  ; ****
_terminate                                ; ****
__ctordtorTerm                            ; ****
exit                                      ; ****
free                                      ; ****
malloc                                    ; ****
strdup                                    ; ****
strpbrk                                   ; ****
```

Figure 8 (Part 2 of 2). SAMPLE3R.DEF - Definition File to Export C Runtime Functions

Note: In this definition file, the `EXPORTS` entry for `selection` is commented out because `selection` is exported explicitly in the code with a **`#pragma export`** statement.

Once you have relinked `SAMPLE03.DLL`, re-create `MAIN03.EXE` so the calls to the VisualAge C++ runtime functions are resolved by dynamically linking to `SAMPLE03.DLL`. A make file, `MAKE03R`, is provided to do this for you.

Note: You must have the Toolkit installed to use the make file.

To re-create `MAIN03.EXE`, at the prompt in the `\IBMCPP\SAMPLES\COMPILER\SAMPLE03` directory under the main VisualAge C++ directory, type:

```
nmake all /f MAKE03R
```

Creating Runtime Library DLLs

To recompile and relink `MAIN03.EXE` yourself:

1. Use the IMPLIB utility to create an import library from `SAMPLE03.DEF`, using the command:

   ```
   IMPLIB SAMPLE03.LIB SAMPLE03.DEF
   ```

2. Compile and link `MAIN03.EXE` with the command:

   ```
   icc /B"/NOE /NOD" MAIN03.C CPPOS300.LIB SAMPLE03.LIB OS2386.LIB
   ```

   Note: If you compiled with the option `/Gn+`, the linker option `/NOD` is not required, but you must recompile all the modules with this option.

   If `MAIN03.OBJ` already exists, you can use the following command to create `MAIN.EXE` by simply relinking:

   ```
   ILINK /NOI /NOE /NOD MAIN03 CPPOS300 SAMPLE03 OS2386;
   ```

After you have performed these steps, copy `SAMPLE03.DLL` to a directory listed in the LIBPATH variable in your CONFIG.SYS file. You can then use the command:

```
MAIN03
```

to run the SAMPLE03 program.

Part 3. Making Your Program International

This section describes internationalization and provides information on VisualAge C++ support for internationalization.

Introduction to Locale

This chapter introduces the concept of internationalization in programming languages and the implementation of internationalization by the use of `locales`. The locale codesets provided with the VisualAge C++ are listed and the default locale set noted. Customization of locales and conversion utilities provided with VisualAge C++ are also discussed.

Internationalization in Programming Languages

Internationalization in programming languages is a concept that comprises

- externally stored *cultural data*,
- a set of *programming tools* to create such cultural data,
- a set of *programming interfaces* to access this data, and
- a set of *programming methods* that enable you to write programs that modify their behavior according to the user's cultural environment, specified during the program's execution.

Elements of Internationalization

A *locale* is a collection of data that encodes information about the cultural environment. The typical elements of cultural environment are as follows:

Native language
The text that the executing program uses to communicate with a user or environment, that is, the natural language of the end user.

Character sets and coded character sets
Maps an *alphabet*, the characters used in a particular language, and a collating sequence onto the set of hexadecimal values (code points) that uniquely identify each character. This mapping creates the coded character set, which is uniquely identified by the character set it encodes, the set of code point values, and the mapping between these two.

Collating and ordering
The relative ordering of characters used for sorting.

Character classification
Determines the type of character (alphabetic, numeric, and so forth) represented by a code point.

Character case conversion
Defines the mapping between uppercase and lowercase characters within a single character set.

Date and time format
Defines the way date and time data (names of weekdays and months; order of month, day, and year, and so forth) are formatted.

Format of numeric and non-numeric numbers
Define the way numbers and monetary units are formatted with commas, decimal points, and so forth.

Note: The VisualAge C++ compiler and library support of internationalization is based on the IEEE POSIX P1003.2 and X/Open Portability Guide standards for global locales and coded character set conversion, with the following exceptions:

- The grouping arguments in the `LC_NUMERIC` and `LC_MONETARY` categories must be strings, not sets of integers.
- The use of the ellipsis (...) in the `LC_COLLATE` category is limited.

For more information about the `LC_NUMERIC`, `LC_MONETARY`, and the `LC_COLLATE` categories, see Appendix C, “Locale Categories” on page 359.

Locales and Localization

Localization is an action that establishes the cultural environment for an application by selecting the active locale. Only one locale can be active at one time, but a program can change the active locale at any time during its execution. The active locale affects the behavior on the locale-sensitive interfaces for the entire program. This is called the *global locale model*.

Locale-Sensitive Interfaces

The VisualAge C++ library products provide many interfaces to manipulate and access locales. You can use these interfaces to write internationalized C programs. The C locale support will also work for C++ programs.

This list summarizes all the VisualAge C++ library functions which affect or are affected by the current locale.

Making Your Program International

Selecting locale
Changing the characteristics of the user's cultural environment by changing the current locale: `setlocale`

Querying locale
Retrieving the locale information that characterizes the user's cultural environment:

Monetary and numeric formatting conventions:
`localeconv`

Date and time formatting conventions:
`localdtconv`

User-specified information:
`nl_langinfo`

Encoding of the variant part of the portable character set:
`getsyntx`

Character set identifier:
`csid`, `wcsid`

Classification of characters:

Single-byte characters:
`isalnum`, `isalpha`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

Wide characters:
`iswalnum`, `iswalpha`, `iswblank`, `iswcntrl`, `iswdigit`, `iswgraph`, `iswlower`, `iswprint`, `iswpunct`, `iswspace`, `iswupper`, `iswxdigit`, `wctype`, `iswctype`

Character case mapping:

Single-byte characters:
`tolower`, `toupper`

Wide characters:
`towlower`, `towupper`

Multibyte character and multibyte string conversion:
`mblen`, `mbrlen`, `mbtowc`, `mbrtowc`, `wctomb`, `wcrtomb`, `mbstowcs`, `mbsrtowcs`, `wcstombs`, `wcsrtombs`, `mbsinit`, `wctob`

String conversions to arithmetic:
`strtod`, `wcstod`, `strtol`, `wcstol`, `strtoul`, `wcstoul`, `atof`, `atoi`, `atol`

String collating:
`strcoll`, `strxfrm`, `wcscoll`, `wcsxfrm`

Character display width:
: `wcswidth`, `wcwidth`

Date, time, and monetary formatting:
: `strftime`, `strptime`, `wcsftime`, `mktime`, `ctime`, `gmtime`, `localtime`, `strfmon`

Formatted input/output:
: `printf` (and family of functions), `scanf` (and family of functions), `vswprintf`, `swprintf`, `swscanf`

Processing regular expressions:
: `regcomp`, `regexec`

Wide character unformatted input/output:
: `fgetwc`, `fgetws`, `fputwc`, `fputws`, `getwc`, `getwchar`, `putwc`, `putwchar`, `ungetwc`

Wide character string handling functions:
: `wcscat`, `wcsncat`, `wcscmp`, `wcsncmp`, `wcscpy`, `wcsncpy`, `wcschr`, `wcscspn`, `wcspbrk`, `wcsspn`, `wcsstok`, `wcsrchr`, `wcslenl`.

Response matching:
: `rpmatch`

Collating elements:
: `ismccollel`, `strtocoll`, `colltostr`, `collequiv`, `collrange`, `collorder`, `cclass`, `maxcoll`, `getmccoll`, `getwmccoll`

Definition of the Default POSIX C Locales

The default `POSIX C` locale is prebuilt into the runtime library.

The `POSIX C` locale is defined as though it was built with a charmap file with a `MB_CUR_MAX` value of 2. The processing of multibyte characters is dependent on the current process codepage. The codeset name for the `nl_langinfo` and `getsyntx` functions is IBM-850.

The following is true of the `POSIX C` locale:

1. It is the default locale.
2. Issuing `setlocale(`*`category`*`, "")` has the following effect:
   - Locale-related environment variables are checked to determine which locales to use to set the *`category`* specified.
   - If no non-null environment variable is present, then it is the equivalent of having issued `setlocale(`*`category`*`, "C")`. That is, the locale chosen is the

C locale defintion, and querying the locale with `setlocale(`*`category`*`, NULL)` returns "C" as the locale name.

The POSIX definition of the C locale is described below, with the IBM extensions LC_SYNTAX and LC_TOD showing their default values.

```
#############
LC_CTYPE
#############
# "alpha" is by default "upper" and "lower"
# "alnum" is by definition "alpha" and "digit"
# "print" is by default "alnum", "punct" and <space> character
# "punct" is by default "alnum" and "punct"

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

digit   <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;\
        <carriage-return>;<space>
cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;\
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
        <IS1>;<DEL>
punct   <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;<commercial-at>;\
        <left-square-bracket>;<backslash>;<right-square-bracket>;\
        <circumflex>;<underscore>;<grave-accent>;\
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;<tilde>
xdigit  <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>;\
        <A>;<B>;<C>;<D>;<E>;<F>;\
        <a>;<b>;<c>;<d>;<e>;<f>
```

```
blank    <space>;\
         <tab>

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
        (<z>,<Z>)

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);\
        (<Z>,<z>)

END LC_CTYPE

#############
LC_COLLATE
#############

order_start
# ASCII Control characters
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
```

```
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
```

```
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>
<H>
<I>
<J>
<K>
<L>
<M>
<N>
<O>
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
```

```
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
order_end

END LC_COLLATE
```

```
#############
LC_MONETARY
#############

int_curr_symbol   ""
currency_symbol   ""
mon_decimal_point ""
mon_thousands_sep ""
mon_grouping      ""
positive_sign     ""
negative_sign     ""
int_frac_digits   -1
frac_digits       -1
p_cs_precedes     -1
p_sep_by_space    -1
n_cs_precedes     -1
n_sep_by_space    -1
p_sign_posn       -1
n_sign_posn       -1

END LC_MONETARY

#############
LC_NUMERIC
#############

decimal_point     "<period>"
thousands_sep     ""
grouping          ""

END LC_NUMERIC

#############
LC_TIME
#############

abday   "<S><u><n>";\
        "<M><o><n>";\
        "<T><u><e>";\
        "<W><e><d>";\
        "<T><h><u>";\
        "<F><r><i>";\
        "<S><a><t>"
```

```
day     "<S><u><n><d><a><y>";\
        "<M><o><n><d><a><y>";\
        "<T><u><e><s><d><a><y>";\
        "<W><e><d><n><e><s><d><a><y>";\
        "<T><h><u><r><s><d><a><y>";\
        "<F><r><i><d><a><y>";\
        "<S><a><t><u><r><d><a><y>"

abmon   "<J><a><n>";\
        "<F><e><b>";\
        "<M><a><r>";\
        "<A><p><r>";\
        "<M><a><y>";\
        "<J><u><n>";\
        "<J><u><l>";\
        "<A><u><g>";\
        "<S><e><p>";\
        "<O><c><t>";\
        "<N><o><v>";\
        "<D><e><c>"

mon     "<J><a><n><u><a><r><y>";\
        "<F><e><b><r><u><a><r><y>";\
        "<M><a><r><c><h>";\
        "<A><p><r><i><l>";\
        "<M><a><y>";\
        "<J><u><n><e>";\
        "<J><u><l><y>";\
        "<A><u><g><u><s><t>";\
        "<S><e><p><t><e><m><b><e><r>";\
        "<O><c><t><o><b><e><r>";\
        "<N><o><v><e><m><b><e><r>";\
        "<D><e><c><e><m><b><e><r>"

# equivalent of AM/PM (%p)
am_pm      "<A><M>";"<P><M>"

# appropriate date and time representation (%c) "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign><b><space><percent-sign><e>\
<space><percent-sign><H><colon><percent-sign><M>\
<colon><percent-sign><S><space><percent-sign><Y>"

# appropriate date representation (%x) "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d><slash><percent-sign><y>"

# appropriate time representation (%X) "%H:%M:%S"
t_fmt      "<percent-sign><M><colon><percent-sign><M><colon><percent-sign><S>"
```

```
# appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon><percent-sign><S>\
<space><percent-sign><p>"

END LC_TIME

#############
LC_MESSAGES
#############

yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"

END LC_MESSAGES
```

LC_syntax and LC_TOD are IBM-extensions to the POSIX C locale definition, their default values are shown here.

```
#############
LC_SYNTAX
#############

backslash     "<backslash>"
right_brace   "<right-brace>"
left_brace    "<left-brace>"
right_bracket "<right-square-bracket>"
left_bracket  "<left-square-bracket>"
circumflex    "<circumflex>"
tilde         "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign   "<number-sign>"
vertical_line "<vertical-line>"
dollar_sign   "<dollar-sign>"
commercial_at "<commercial-at>"
grave_accent  "<grave-accent>"

END LC-SYNTAX
```

```
#############
LC_TOD
#############

timezone_difference  0
timezone_name        ""
daylight_name        ""
start_month          0
end_month            0
start_week           0
end_week             0
start_day            0
end_day              0
start_time           0
end_time             0
shift                0

END LC_TOD
```

Differences Between SAA C and POSIX C Locales

The incompatibilities between the `POSIX C` locale of the current VisualAge C++ (Version 3) and the `SAA C` locale available with previous versions of VisualAge C++ are as follows:

LC_TIME category

date/time format
The date/time format in the previous `C` locale is "%y/%m/%d %I:%M:%S". In the `POSIX` locale, it is "%a %b %d %H %M %S %Y".

time format
The time format in the previous `C` locale is "%I:%M:%S". In the `POSIX` locale, it is "%H:%M:%S".

am/pm strings
The am/pm strings in the previous `C` locale are "am" and "pm". In the `POSIX` locale, they are "AM" and "PM".

Customizing a Locale

This section describes how you can create your own locales, based on the locale definition files supplied by IBM. The information in this chapter applies to the format of locales based on the LOCALDEF utility.

In this example you will build a locale named `TEXAN` using the `charmap` file representing the IBM-437 encoded character set. The locale is derived from the locale representing the English language and the cultural conventions of the United States.

1. Determine the source of the locale you are going to use. In this case, it is the locale for the English language in the United States, the source for which is `EN_US\IBM-437.LOC`.
2. Copy the selected file (`EN_US\IBM-437.LOC`) from the source directory to your own directory and rename it. For example, assuming you are in the C Set ++ locale directory,

   ```
   c:\ibmcpp\locale
   ```

   you would type the following:

   ```
      copy EN_US\IBM-437.LOC c:\ibmcpp\locale\fred\TEXAN.LOC
   ```
3. In your new file, change the locale variables to the desired values. For example, change

   ```
   d_t_fmt "%a %b %e %H:%M:%S %Z %Y
   ```

   to

   ```
   d_t_fmt "Howdy Pardner %a %b %e %H:%M:%S %Z %Y"
   ```
4. Generate a new locale load module using the LOCALDEF utility, then place the produced module in the directory where your locale load modules are located. Of course, this directory must be specified in the LOCPATH variable.

   ```
   localdef /f ibm-437.cm /i texan.loc texan.lcl
   ```

   See the *User's Guide* for detailed information about the syntax of the LOCALDEF utility.

Using the Customized Locale

The customized locale is now ready to be used in calls made by the `setlocale` function in VisualAge C++ application code, such as:

```
setlocale(LC_ALL, "texan");
```

Referring Explicitly to a Customized Locale

Here is a program with an explicit reference to the `TEXAN` locale.

```
/* this example shows how to get the local time formatted by the */
/* current locale */

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

int main(void){
    char dest[80];
    int ch;
    time_t temp;
    struct tm *timeptr;
    temp = time(NULL);
    timeptr = localtime(&temp);
    /* Fetch default locale name */
    printf("Default locale is %s\n",setlocale(LC_ALL,"C"));
    ch = strftime(dest,sizeof(dest)-1,
      "Local C datetime is %c", timeptr);
    printf("%s\n",  dest);

    /* Set new Texan locale name */
    printf("New locale is %s\n", setlocale(LC_ALL,"Texan"));
    ch = strftime(dest,sizeof(dest)-1,
      "Texan datetime is %c ", timeptr);
    printf("%s\n", dest);

    return(0);
}
```

Figure 9. Referring Explicitly to a Customized Locale

Compile and run the above program. The output should be similar to:

```
Default locale is "C"
Local C datetime is Fri Aug 20 14:58:12  1993
New locale is TEXAN
Texan datetime is Howdy Pardner Fri Aug 20 14:58:12  1993
```

Using Environment Variables to Select a Locale

You can use environment variables to specify the names of locale categories. You must call `setlocale` regardless of environmental variable settings. However, if you call `setlocale.` without specifying the locale argument, the locale is changed according to environmental variables.

Here is an example:

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

int main(void){
  setlocale(LC_ALL,""));
  printf("default -"-" locale = %s-n", setlocale(LC_ALL,"NULL"));
  _putenv("LC_ALL=TEXAN");
  setlocale(LC_ALL,""));
  printf("Default -"-" locale = %s-n", setlocale(LC_ALL,"NULL"));
  return(0);

}
```

Figure 10. Using Environment Variables to Select a Locale

If you run the program above, you can expect the following result:

```
Default "" locale = C
Default "" locale = TEXAN
```

Note: Specifying NULL as a locale name in a `setlocale` call means 'query current locale'

In the example above, the default NULL locale returns C because the value of `LC_ALL` does not affect the current locale until the next `setlocale(LC_ALL,"")` is done. When this call is made, the `LC_ALL` environment variable will be used and the locale will be set to `TEXAN`.

For more information about setting environment variables, see Chapter 1, "Setting Runtime Environment Variables" on page 3.

The names of the environment variables match the names of the locale categories:

- `LC_ALL`
- `LC_COLLATE`
- `LC_CTYPE`
- `LANG`
- `LC_MESSAGES`
- `LC_MONETARY`
- `LC_NUMERIC`
- `LC_TIME`
- `LC_TOD`
- `LC_SYNTAX`

See the *C Library Reference* for information about `setlocale`.

Code Set Conversion Utilities

This section describes the code set conversion utilities supported by the VisualAge C++ compiler. These utilities are as follows:

ICONV utility
Converts a file from one code set encoding to another.

`iconv` **functions**
Perform code set translation. These functions are `iconv_open`, `iconv`, and `iconv_close`. They are used by the ICONV utility and may be called from any VisualAge C++ program requiring code set translation.

GENXLT utility
Generates a translation table for use by the ICONV utility and `iconv` functions.

See the *User's Guide* for descriptions of the GENXLT and ICONV utilities, and the *C Library Reference* for descriptions of the `iconv` functions.

The GENXLT Utility

The GENXLT utility reads a source translation file from a specified input file and writes the compiled version to a specified output file. If you do not specify an input file or you do not specify an output file, GENXLT uses standard input (**stdin**) and standard output (**stdout**), respectively. The source translation file contains directives that are acted upon by the GENXLT utility to produce the compiled version of the translation table.

For more information on the GENXLT tool, see the *User's Guide*.

The ICONV Utility

The ICONV utility reads characters from the input file, converts them from one coded character set definition to another, and writes them to the output file.

For more information on the ICONV utility, see the *User's Guide*.

Code Conversion Functions

The `iconv_open`, `iconv`, and `iconv_close` library functions can be called from C language source to initialize and perform the characters conversions from one character set encoding to another.

For more information on these functions, see the *C Library Reference*.

Code Set Converters Supplied

The code set converters that are provided with VisualAge C++ are as follows:

- Code set converters between the `Latin-1` and non-`Latin-1` code pages and code page IBM-850.
- Code set converters to convert to and from IBM-850, IBM-1047, and ISO8859-1.
- Code set converters to convert between ISO8859-7 and ISO8859-9 and the applicable ASCII code page.
- Code set converters between the Japanese code pages.

The code set converters are provided either as tables built by the GENXLT utility or as functions inside `IBMCCONV.DLL`. The input source to the GENXLT utility is also provided.

The code set converters use the "Enforced subset match" method for characters that are in the input code set but are not in the output code set. All characters not in the output code set are replaced by the `SUB` character, which is `0x3F` in `EBCDIC` and `0x1A` in ASCII.

The following table lists the code set converters supplied with VisualAge C++:

FromCode	ToCode	Function/Pathname
EBCDIC Codesets to ASCII Codesets		
IBM-037	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-037\IBM-850.XLT
IBM-273	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-273\IBM-850.XLT
IBM-274	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-274\IBM-850.XLT
IBM-275	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-274\IBM-850.XLT
IBM-277	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-277\IBM-850.XLT
IBM-278	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-278\IBM-850.XLT
IBM-280	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-280\IBM-850.XLT
IBM-281	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-281\IBM-850.XLT
IBM-282	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-282\IBM-850.XLT
IBM-284	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-284\IBM-850.XLT
IBM-285	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-285\IBM-850.XLT
IBM-297	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-297\IBM-850.XLT
IBM-500	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-500\IBM-850.XLT
IBM-871	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-871\IBM-850.XLT
IBM-875	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-875\IBM-850.XLT
IBM-930	IBM-932	IBM-930_IBM-932
IBM-939	IBM-932	IBM-939_IBM-932
IBM-1026	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-1026\IBM-850.XLT
IBM-1047	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-1047\IBM-850.XLT

FromCode	ToCode	Function/Pathname
ASCII Codesets to EBCDIC Codesets		
IBM-850	IBM-037	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-037.XLT
IBM-850	IBM-273	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-273.XLT
IBM-850	IBM-274	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-274.XLT
IBM-850	IBM-275	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-275.XLT
IBM-850	IBM-277	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-277.XLT
IBM-850	IBM-278	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-278.XLT
IBM-850	IBM-280	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-280.XLT
IBM-850	IBM-281	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-281.XLT
IBM-850	IBM-282	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-282.XLT
IBM-850	IBM-284	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-284.XLT
IBM-850	IBM-285	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-285.XLT
IBM-850	IBM-297	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-297.XLT
IBM-850	IBM-500	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-500.XLT
IBM-850	IBM-871	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-871.XLT
IBM-850	IBM-875	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-875.XLT
IBM-850	IBM-1026	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-1026.XLT
IBM-932	IBM-930	IBM-932_IBM-930
IBM-932	IBM-939	IBM-932_IBM-939

FromCode	ToCode	Function/Pathname
ASCII Codesets to ASCII Codesets		
IBM-437	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-437\IBM-850.XLT
IBM-852	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-852\IBM-850.XLT
IBM-857	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-857\IBM-850.XLT
IBM-860	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-860\IBM-850.XLT
IBM-861	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-861\IBM-850.XLT
IBM-863	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-863\IBM-850.XLT
IBM-865	IBM-850	\IBMC\LOCALE\ICONVTAB\IBM-865\IBM-850.XLT
IBM-850	IBM-437	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-437.XLT
IBM-850	IBM-852	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-852.XLT
IBM-850	IBM-857	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-857.XLT
IBM-850	IBM-860	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-860.XLT
IBM-850	IBM-861	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-861.XLT
IBM-850	IBM-863	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-863.XLT
IBM-850	IBM-865	\IBMC\LOCALE\ICONVTAB\IBM-850\IBM-865.XLT
EBCDIC Codesets to EUCJP Codeset		
IBM-eucJP	IBM-930	IBM-eucJP_IBM-930
IBM-eucJP	IBM-939	IBM-eucJP_IBM-939
IBM-930	IBM-eucJP	IBM-930_IBM-eucJP
IBM-939	IBM-eucJP	IBM-939_IBM-eucJP
ISO Codesets to ASCII Codesets		
ISO8859-1	IBM-850	\IBMC\LOCALE\ICONVTAB\ISO88591\IBM-850.XLT
ISO8859-9	IBM-857	\IBMC\LOCALE\ICONVTAB\ISO88599\IBM-857.XLT
IBM-850	ISO8859-1	\IBMC\LOCALE\ICONVTAB\IBM-850\ISO88591.XLT
IBM-857	ISO8859-9	\IBMC\LOCALE\ICONVTAB\IBM-857\ISO88599.XLT

The following code set converters are also supplied. These converters are used by the code set converters between the codesets IBM-930, IBM-932, IBM-939, and IBM-eucJP.

FromCode	**ToCode**	**Function/Pathname**
IBM-290	IBM-932	\IBMC\LOCALE\ICONVTAB\IBM-290\IBM-932.XLT
IBM-290	IBM-eucJP	IBM-290_IBM-eucJP
IBM-300	IBM-932	IBM-300_IBM-932
IBM-300	IBM-eucJP	IBM-300_IBM-eucJP
IBM-932	IBM-290	\IBMC\LOCALE\ICONVTAB\IBM-932\IBM-290.XLT
IBM-932	IBM-300	IBM-932_IBM-300
IBM-932	IBM-1027	\IBMC\LOCALE\ICONVTAB\IBM-932\IBM-1027.XLT
IBM-1027	IBM-932	\IBMC\LOCALE\ICONVTAB\IBM-1027\IBM-932.XLT
IBM-1027	IBM-eucJP	IBM-1027_IBM-eucJP
IBM-eucJP	IBM-290	IBM-eucJP_IBM-290
IBM-eucJP	IBM-300	IBM-eucJP_IBM-300
IBM-eucJP	IBM-1027	IBM-eucJP_IBM-1027

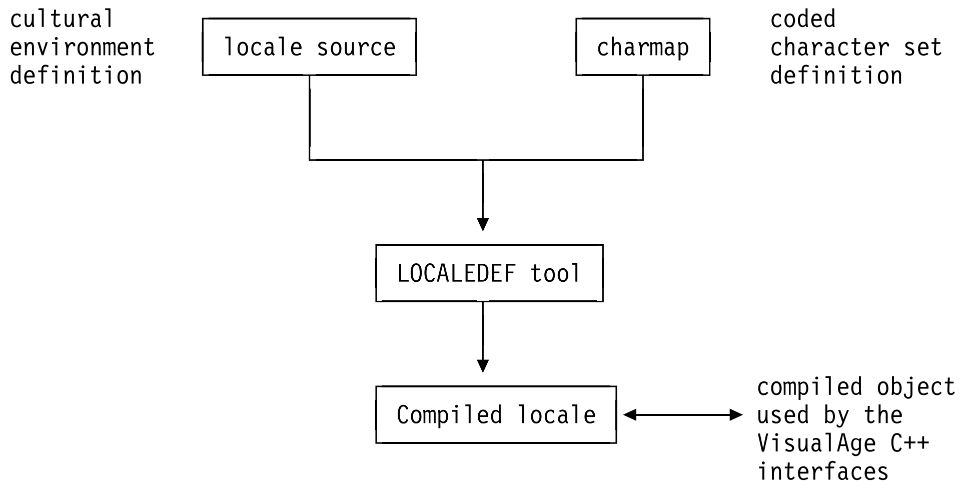
8 Building a Locale

Cultural information is encoded in the locale source file using the locale definition language. One locale source file characterizes one cultural environment.

The locale source file is processed by the locale compilation tool, called the LOCALDEF tool. See the *User's Guide* for information on using this tool.

To enhance portability of the locale source files, certain information related to the character sets can be encoded using the symbolic names of characters. The mapping between the symbolic names and the characters they represent and its associated hexadecimal value is defined in the *character set description file* or `charmap` file.

The conceptual model of the locale build process is presented below:


cultural environment definition
locale source
charmap
coded character set definition
LOCALEDEF tool
Compiled locale
compiled object used by the VisualAge C++ interfaces

Using the charmap File

The `charmap` file defines a mapping between the symbolic names of characters and the hexadecimal values associated with the character in a given coded character set. Optionally, it can provide the alternate symbolic names for characters. Characters in the locale source file can be referred to by their symbolic names or alternate symbolic names, thereby allowing for writing generic locale source files independent of the encoding of the character set they represent.

Each `charmap` file must contain at least the definition of the portable character set and the character symbolic names associated with each character. The characters in the portable character set and the corresponding symbolic names, and optional alternate symbolic names, are defined in Figure 11 on page 114.

Figure 11 (Page 1 of 4). Characters in portable character set and corresponding symbolic names

Symbolic Name	Alternate Name	Character
`<NUL>`		
`<tab>`	`<SE10>`	
`<vertical-tab>`	`<SE12>`	
`<form-feed>`	`<SE13>`	
`<carriage-return>`	`<SE14>`	
`<newline>`	`<SE11>`	
`<backspace>`	`<SE09>`	
`<alert>`	`<SE08>`	
`<space>`	`<SP01>`	
`<period>`	`<SP11>`	.
`<less-than-sign>`	`<SA03>`	<
`<left-parenthesis>`	`<SP06>`	(
`<plus-sign>`	`<SA01>`	+
`<ampersand>`	`<SM03>`	&
`<right-parenthesis>`	`<SP07>`	)
`<semicolon>`	`<SP14>`	;
`<hyphen>`	`<SP10>`	-
`<hyphen-minus>`	`<SP10>`	-
`<slash>`	`<SP12>`	/
`<solidus>`	`<SP12>`	/
`<comma>`	`<SP08>`	,
`<percent-sign>`	`<SM02>`	%
`<underscore>`	`<SP09>`	_
`<low-line>`	`<SP09>`	_
`<greater-than-sign>`	`<SA05>`	>
`<question-mark>`	`<SP15>`	?
`<colon>`	`<SP13>`	:

Figure 11 (Page 2 of 4). Characters in portable character set and corresponding symbolic names

Symbolic Name	Alternate Name	Character
<apostrophe>	<SP05>	'
<equals-sign>	<SA04>	=
<quotation-mark>	<SP04>	"
<a>	<LA01>	a
<b>	<LB01>	b
<c>	<LC01>	c
<d>	<LD01>	d
<e>	<LE01>	e
<f>	<LF01>	f
<g>	<LG01>	g
<h>	<LH01>	h
<i>	<LI01>	i
<j>	<LJ01>	j
<k>	<LK01>	k
<l>	<LL01>	l
<m>	<LM01>	m
<n>	<LN01>	n
<o>	<LO01>	o
<p>	<LP01>	p
<q>	<LQ01>	q
<r>	<LR01>	r
<s>	<LS01>	s
<t>	<LT01>	t
<u>	<LU01>	u
<v>	<LU01>	v
<w>	<LW01>	w
<x>	<LX01>	x
<y>	<LY01>	y
<z>	<LZ01>	z

Figure 11 (Page 3 of 4). Characters in portable character set and corresponding symbolic names

Symbolic Name	Alternate Name	Character
<A>	<LA02>	A
<B>	<LB02>	B
<C>	<LC02>	C
<D>	<LD02>	D
<E>	<LE02>	E
<F>	<LF02>	F
<G>	<LG02>	G
<H>	<LH02>	H
<I>	<LI02>	I
<J>	<LJ02>	J
<K>	<LK02>	K
<L>	<LL02>	L
<M>	<SM02>	M
<N>	<LN02>	N
<O>	<LO02>	O
<P>	<LP02>	P
<Q>	<LQ02>	Q
<R>	<LR02>	R
<S>	<LS02>	S
<T>	<LT02>	T
<U>	<LU02>	U
<V>	<LV02>	V
<W>	<LW02>	W
<X>	<LX02>	X
<Y>	<LY02>	Y
<Z>	<LZ02>	Z
<zero>	<ND10>	0
<one>	<ND01>	1
<two>	<ND02>	2

Figure 11 (Page 4 of 4). Characters in portable character set and corresponding symbolic names

Symbolic Name	Alternate Name	Character
<three>	<ND03>	3
<four>	<ND04>	4
<five>	<ND05>	5
<six>	<ND06>	6
<seven>	<ND07>	7
<eight>	<ND08>	8
<nine>	<ND09>	9
<vertical-line>	<SM13>	|
<exclamation-mark>	<SP02>	!
<dollar-sign>	<SC03>	$
<circumflex>	<SD15>	^
<circumflex-accent>	<SD15>	^
<grave-accent>	<SD13>	`
<number-sign>	<SM01>	#
<commercial-at>	<SM05>	@
<tilde>	<SD19>	~
<left-square-bracket>	<SM06>	[
<right-square-bracket>	<SM08>	]
<left-brace>	<SM11>	{
<left-curly-bracket>	<SM11>	{
<right-brace>	<SM14>	}
<right-curly-bracket>	<SM14>	}
<backslash>	<SM07>	\
<reverse-solidus>	<SM07>	\

The portable character set is the basis for the syntactic and semantic processing of the LOCALDEF tool, and for most of the utilities and functions that access the locale object files. Therefore the portable character set must always be defined.

The `charmap` file is divided into two main sections:

1. the character symbolic name to hexidecimal mapping section, or `CHARMAP`

2. the character symbolic name to character set identifier section, or `CHARSETID`

The following definitions can precede the two sections listed above. Each consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more `<blank>`s, followed by the value to be assigned to the symbol.

`<code_set_name>`
: The string literal containing the name of the coded character set name

`<mb_cur_max>`
: the maximum number of bytes in a multibyte character which can be set to a value of either 1 or 2. If it is 1, each character in the character set defined in this `charmap` is encoded by a one-byte value. If it is 2, each character in the character set defined in this `charmap` is encoded by a one- or two-byte value. If it is not specified, the default value of 1 is assumed. If a value of other than 1 or 2, is specified, a warning message is issued and the default value of 1 is assumed.

`<mb_cur_min>`
: The minimum number of bytes in a multibyte character. Can be set to 1 only. If a value of other than 1 is specified, a warning message is issued and the default value of 1 is assumed.

`<escape_char>`
: Specifies the escape character that is used to specify hexadecimal or octal notation for numeric values. It defaults to the hexadecimal value `0x5C`, which represents the `\` character in the coded character set IBM-850.

`<comment_char>`
: Denotes the character chosen to indicate a comment within a charmap file. It defaults to the hexadecimal value `0x23`, which represents the `#` character in the coded character set IBM-850.

The CHARMAP Section

The `CHARMAP` section defines the values for the symbolic names representing characters in the coded character set. Each `charmap` file must define at least the portable character set. The character symbolic names or alternate symbolic names (or both) must be be used to define the portable character set. These are shown in Figure 11 on page 114.

Additional characters can be defined by the user with symbolic character names.

The `CHARMAP` section starts with the line containing the keyword `CHARMAP`, and ends with the line containing the keywords `END CHARMAP`. `CHARMAP` and `END CHARMAP` must both start in column one.

The character set mapping definitions are all the lines between the first and last lines of the `CHARMAP` section.

The formats of the character set mappings for this section are as follows:

```
"%s %s %s\n", <symbolic-name>, <encoding>, <comments>
"%s...%s %s %s\n", <symbolic-name>, <symbolic-name>, <encoding>, <comments>
```

The first format defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters with visible glyphs, enclosed between angle brackets.

A character following an escape character is interpreted as itself; for example, the sequence `<\\\>>` represents the symbolic name `\>` enclosed within angle brackets, where the backslash (\) is the escape character.

The second format defines a group of symbolic names associated with a range of values. The two symbolic names are comprised of two parts, a prefix and suffix. The prefix consists of zero or more non-numeric invariant visible glyph characters and is the same for both symbolic names. The suffix consists of a positive decimal integer. The suffix of the first symbolic name must be less than or equal to the suffix of the second symbolic name. As an example, `<j0101>...<j0104>` is interpreted as the symbolic names `<j0101>,<j0102>,<j0103>,<j0104>`. The common prefix is `'j'` and the suffixes are `'0101'` and `'0104'`.

The encoding part can be written in one of two forms:

```
<escape-char><number>                        (single byte value)
<escape-char><number><escape-char><number>  (double byte value)
```

The number can be written using octal, decimal, or hexadecimal notation. Decimal numbers are written as a `'d'` followed by 2 or 3 decimal digits. Hexadecimal numbers are written as an `'x'` followed by 2 hexadecimal digits. An octal number is written with 2 or 3 octal digits. As an example, the single byte value `x1F` could be written as `'\37'`, `'\x1F'`, or `'\d31'`. The double byte value of `x1A1F` could be written as `'\32\37'`, `'\x1A\x1F'`, or `'\d26\d31'`.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent names defined by the range have encoding values in increasing order.

When constants are concatenated for multibyte character values, they must be of the same type, and are interpreted in byte order from first to last with the least significant byte of the multibyte character specified by the last constant. For example, the following line:

```
<j0101>...<j0104>           \d129\d254
```

would be interpreted as follows:

```
<j0101>                           \d129\d254
<j0102>                           \d129\d255
<j0103>                           \d130\d0
<j0104>                           \d130\d1
```

The CHARSETID Section

The character set identifier section of the `charmap` file maps the symbolic names defined in the `CHARMAP` section to a character set identifier.

Note: The two functions `csid` and `wcsid` query the locales and return the character set identifier for a given character. This information is not currently used by any other library function.

The `CHARSETID` section starts with a line containing the keyword `CHARSETID`, and ends with the line containing the keywords `END CHARSETID`. Both `CHARSETID` and `END CHARSETID` must begin in column 1. The lines between the first and last lines of the `CHARSETID` section define the character set identifier for the defined coded character set.

The character set identifier mappings are defined as follows:

```
"%s %c", <symbolic-name>, <value>
"%c %c", <value>, <value>
"%s...%s %c", <symbolic-name>, <symbolic-name>, <value>
"%c...%c %c", <value>, <value>, <value>
"%s...%c %c", <symbolic-name>, <value>, <value>
"%c...%s %c", <value>, <symbolic-name>, <value>
```

The individual characters are specified by the symbolic name or the value. The group of characters are specified by two symbolic names or by two numeric values (or combination) separated by an ellipsis (...). The interpretation of ranges of values is the same as specified in the `CHARMAP` section. The character set identifier is specified by a numeric value.

Locale Source Files

Locales are defined through the specification of a locale definition file. The locale definition contains one or more distinct locale category source definitions and not more than one definition of any category. Each category controls specific aspects of the cultural environment. A category source definition is either the explicit definition of a category or the `copy` directive, which indicates that the category definition should be copied from another locale definition file.

The definition file is composed of an optional definition section for the escape and comment characters to be used, followed by the category source definitions. Comment lines and blank lines can appear anywhere in the locale definition file. If the escape and comment characters are not defined, default code points are used (`x5C` for the escape character and `x23` for the comment character, respectively). The definition section consists of the following optional lines:

```
escape_char     <character>
comment_char    <character>
```

where `<character>` in both cases is a single-byte character to be used, for example:

```
escape_char     /
```

defines the escape character in this file to be `'/'` (the `<slash>` character).

Locale definition files passed to the `localedef` utility are assumed to be in coded character set IBM-850.

Each category source definition consists of a category header, a category body, and a category trailer, in that order.

category header

consists of the keyword naming the category. Each category name starts with the characters `LC_` The following category names are supported: `LC_CTYPE, LC_COLLATE, LC_NUMERIC, LC_MONETARY, LC_TIME, LC_MESSAGES, LC_TOD`, and `LC_SYNTAX`.

The `LC_TOD` and `LC_SYNTAX` categories, if present, must be the last two categories in the locale definition file.

category body

consists of one or more lines describing the components of the category. Each component line has the following format:

```
<identifer>   <operand1>
<identifer>   <operand1>;<operand2>;...;<operandN>
```

`<identifier>` is a keyword that identifies a locale element, or a symbolic name that identifies a collating element. `<operand>` is a character, collating element, or string literal. Escape sequences can be specified in a string literal using the `<escape_character>`. If multiple operands are specified, they must be separated by semicolons. White space can be before and after the semicolons.

category trailer
consists of the keyword END followed by one or more <blank>s and the category name of the corresponding category header.

Here is an example of locale source containing the header, body, and trailer:

```
# Here is a simple locale definition file consisting of one
# category source definition, LC_CTYPE.

LC_CTYPE
upper <A>;...;<Z>
END LC_CTYPE
```

You do not have to define each category. Where category definitions are absent from the locale source, default definitions are used.

In each category the keyword copy followed by a string specifies the name of an existing locale to be used as the source for the definition of this category. The compiler searches for a specified existing locale as follows:

1. If you specify a path, the compiler searches that path.
2. If you specify a file name but no path, the compiler searches the current directory.
3. If you specify a file name but no path and the file is not in the current directory, the compiler searches the paths that you specified in the DPATH environment variable.

If the locale is not found, an error is reported and no locale output is created.

You can continue a line in a locale definition file by placing an escape character as the last character on the line. This continuation character is discarded from the input. Even though there is no limitation on the length of each line, for portability reasons it is suggested that each line be no longer than 2048 characters (bytes). There is no limit on the accumulated length of a continued line. You cannot continue comment lines on a subsequent line by using an escaped <newline>.

Individual characters, characters in strings, and collating elements are represented using symbolic names, as defined below. Characters can also be represented as the characters themselves, or as octal, hexadecimal, or decimal constants. If you use non-symbolic notation, the resultant locale definition file may not be portable among systems and environments. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; if you use it to represent itself, you must precede it with the escape character.

The following rules apply to the character representation:

1. A character can be represented by a symbolic name, enclosed within angle brackets. The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the `charmap` file. The symbolic name is replaced by the character value determined from the value associated with the symbolic name in the `charmap` file.

   The use of a symbolic name not found in the `charmap` file constitutes an error, unless the name is in the category `LC_CTYPE` or `LC_COLLATE`, in which case it constitutes a warning. Use of the escape character or right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character. For example:

`<c>;<c-cedilla>`	specifies two characters whose symbolic names are "`c`" and "`c-cedilla`"
`"<M><a><y>"`	specifies a 3-character string composed of letters represented by symbolic names "`M`", "`a`", and "`y`".
`"<a><\>>"`	specifies a 2-character string composed of letters represented by symbolic names "`a`" and "`>`" (assuming the escape character is `\`)

2. A character can represent itself. Within a string, the double quotation mark, the escape character, and the left angle bracket must be escaped (preceded by the escape character) to be interpreted as the characters themselves. For example:

`c`	`'c'` character represented by itself
`"may"`	represents a 3-character string, each character within the string represented by itself
`"###"#>"`	represents the three character long string "`#">`", where the escape character is defined as `#`.

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value.

   For example:

   ```
   \131 "\212\129\168" \16\66\193\17
   ```

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character, followed by an `x`, followed by two or more hexadecimal digits. Each constant represents a byte value.

   Example: `\x83 "\xD4\x81\xA8"`

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a `d` followed by two or more decimal digits. Each constant represents a byte value.

Example: `\d131 "\d212\d129\d168" \d14\d66\d193\d15`

Multibyte characters can be represented by concatenating constants specified in byte order with the last constant specifying the least significant byte of the character.

Using the LOCALDEF Utility

The locale objects or locales are generated using the LOCALDEF utility. The LOCALDEF utility:

1. Reads the *locale definition file*.
2. Resolves all the character symbolic names to the values of characters defined in the specified *character set definition file*.
3. Produces a VisualAge C++ source file.
4. Compiles the source file using the VisualAge C++compiler and links the object file to produce a locale module.

The locale module can be loaded by the `setlocale` function and then accessed by the VisualAge C++ functions that are sensitive to the cultural information, or that can query the locales. For a list of all the library functions sensitive to locale, see "Locale-Sensitive Interfaces" on page 92. For detailed information on how to invoke the LOCALDEF utility, see the *User's Guide*.

Locale Naming Conventions

The `setlocale` library function that selects the active locale maps the descriptive locale name into the name of the locale object before loading the locale and making it accessible.

In VisualAge C++ programs, the locale modules are referred to by descriptive locale names. The locale names themselves are not case sensitive. They follow these conventions:

```
<Language>_<Territory>.<Codeset>
```

Where:

Language
: is a two-letter abbreviation for the language name. The abbreviations come from the ISO 639 standard.

Territory
: is a two-letter abbreviation for the territory name. The abbreviation comes from the ISO 3166 standard.

Codeset
: is the name registered by the MIT X Consortium that identifies the registration authority that owns the specific encoding.

  A modifier may be added to the registered name but is not required. The modifier is of the form `@modifier` and identifies the coded character set as defined by that registration authority.

  Note: On FAT file systems, the modifier cannot be used as it causes the filename to exceed the 8 character FAT filename limit.

The `Codeset` parts are optional. If they are not specified, `Codeset` defaults to `IBM-nnn`, where `nnn` is the current code page. (The modifier portion defaults to nothing.)

The locale name parameter is used to locate the locale as follows:

- If the locale name contains the drive letter (for example, `C:`) or the backslash character, it is a fully-qualified name (such as `\IBMCPP\LOCALE\FRED\IBM-850`). The name must specify the name of a DLL that contains the locale. If you do not specify the extension `.LCL`, it is appended to the name.
- If the locale name does not contain a drive letter or the backslash character, it is not a fully-qualified name. The `setlocale` function tries to load the locale from the current directory, or from the directories that you specified in the `LOCPATH` environment variable. The specified paths are searched to find the DLL that contains the locale.

The locale name parameter is processed to produce a filename suitable for use with the FAT file system:

1. The locale name parameter is separated into two parts, `language_territory` and `codeset`.
2. If you specify the codeset name, the locale name is built as:

   ```
   language_territory\codeset.LCL
   ```
3. If you do not specify the codeset name, the locale name is built as `language_territory.LCL`.
4. If the locale cannot be found, the `DosQueryCp` function determines the current codepage. The codeset name is built as `IBM-`*`nnn`*, where *`nnn`* is the current codepage.

   The locale name is then built as:

   ```
   language_territory\codeset.LCL
   ```

The following locale names are provided:

Figure 12 (Page 1 of 3). Compiled locales supplied with VisualAge C++

Locale Name as in setlocale() argument	Language	Country	Codeset	Locale Module Name
C				
DA_DK.IBM-850	Danish	Denmark	IBM-850	DA_DK\IBM-850
DA_DK.IBM-865	Danish	Denmark	IBM-865	DA_DK\IBM-865
DE_CH.IBM-437	Deutsch (German)	Switzerland	IBM-437	DE_CH\IBM-437
DE_CH.IBM-850	Deutsch (German)	Switzerland	IBM-850	DE_CH\IBM-850
DE_DE.IBM-437	Deutsch (German)	Germany	IBM-437	DE_DE\IBM-437
DE_DE.IBM-850	Deutsch (German)	Germany	IBM-850	DE_DE\IBM-850
EN_GB.IBM-437	English	United Kingdom	IBM-437	EN_GB\IBM-437
EN_GB.IBM-850	English	United Kingdom	IBM-850	EN_GB\IBM-850
EN_JP.IBM-437	English	Japan	IBM-437	EN_JP\IBM-437
EN_JP.IBM-850	English	Japan	IBM-850	EN_JP\IBM-850
EN_US.IBM-437	English	United States	IBM-437	EN_US\IBM-437
EN_US.IBM-850	English	United States	IBM-850	EN_US\IBM-850
ES_ES.IBM-437	Español (Spanish)	Spain	IBM-437	EN_ES\IBM-437
ES_ES.IBM-850	Español (Spanish)	Spain	IBM-850	EN_ES\IBM-850
FI_FI.IBM-437	Finnish	Finland	IBM-437	FI_FI\IBM-437
FI_FI.IBM-850	Finnish	Finland	IBM-850	FI_FI\IBM-850

Figure 12 (Page 2 of 3). Compiled locales supplied with VisualAge C++

Locale Name as in setlocale() argument	Language	Country	Codeset	Locale Module Name
FR_BE.IBM-437	French	Belgium	IBM-437	FR_BE\IBM-437
FR_BE.IBM-850	French	Belgium	IBM-850	FR_BE\IBM-850
FR_CA.IBM-850	French	Canada	IBM-850	FR_CA\IBM-850
FR_CA.IBM-863	French	Canada	IBM-863	FR_CA\IBM-863
FR_CH.IBM-437	French	Switzerland	IBM-437	FR_CH\IBM-437
FR_CH.IBM-850	French	Switzerland	IBM-850	FR_CH\IBM-850
FR_FR.IBM-437	French	France	IBM-437	FR_FR\IBM-437
FR_FR.IBM-850	French	France	IBM-850	FR_FR\IBM-850
IS_IS.IBM-850	Íslensk (Icelandic)	Iceland	IBM-850	IS_IS\IBM-850
IS_IS.IBM-861	Íslensk (Icelandic)	Iceland	IBM-861	IS_IS\IBM-861
IT_IT.IBM-437	Italian	Italy	IBM-437	IT_IT\IBM-437
IT_IT.IBM-850	Italian	Italy	IBM-850	IT_IT\IBM-850
JA_JP.IBM-932	Japanese	Japan	IBM-932	JA_JP\IBM-932
NL_BE.IBM-437	Nederlands (Dutch)	Belgium	IBM-437	NL_BE\IBM-437
NL_BE.IBM-850	Nederlands (Dutch)	Belgium	IBM-850	NL_BE\IBM-850
NL_NL.IBM-437	Nederlands (Dutch)	Netherlands	IBM-437	NL_NL\IBM-437
NL_NL.IBM-850	Nederlands (Dutch)	Netherlands	IBM-850	NL_NL\IBM-850

Figure 12 (Page 3 of 3). Compiled locales supplied with VisualAge C++

Locale Name as in setlocale() argument	Language	Country	Codeset	Locale Module Name
NO_NO.IBM-850	Norwegian	Norway	IBM-850	NO_NO\IBM-850
NO_NO.IBM-865	Norwegian	Norway	IBM-865	NO_NO\IBM-865
PT_PT.IBM-850	Portuguese	Portugal	IBM-850	PT_PT\IBM-850
PT_PT.IBM-860	Portuguese	Portugal	IBM-860	PT_PT\IBM-860
SV_SE.IBM-437	Svensk (Swedish)	Sweden	IBM-437	SV_SE\IBM-437
SV_SE.IBM-850	Svensk (Swedish)	Sweden	IBM-850	SV_SE\IBM-850
TR_TR.IBM-857	Turkish	Turkey	IBM-857	TR_TR\IBM-857

The exceptions to the rule above are the following special locale names, which are already recognized:

- `C`
- `POSIX`
- `GERM`
- `FRAN`
- `UK`
- `ITAL`
- `SPAI`
- `USA`
- `JAPN`
- `JAP2`
- `JAP3`
- `CDEF` which is compatible with C locale from the previous release.

You can use the following macros, defined in the `locale.h` header file, as synonyms for the special locale names above.

Macro	Locale
`LC_C`	`"C"`
`LC_C_GERMANY`	`"GERM"`
`LC_C_FRANCE`	`"FRAN"`
`LC_C_UK`	`"UK"`
`LC_C_ITALY`	`"ITAL"`
`LC_C_SPAIN`	`"SPAI"`
`LC_C_USA`	`"USA"`
`LC_C_JAPAN`	`"JAPN"`
`LC_C_JAPAN2`	`"JAP2"`
`LC_C_JAPAN3`	`"JAP3"`

Building a Locale

Part 4. Advanced Topics

This part describes some of the advanced features of the VisualAge C++ compiler.

Advanced Topics

Using Templates in C++ Programs

Templates may be used in C++ to declare and define classes, functions, and static data members of template classes. The C++ language describes the syntax and meaning of each kind of template. Each particular compiler, however, determines the mechanism that controls when and how often a template is expanded.

VisualAge C++ offers several alternative organizations with a range of convenience and compile performance to meet the needs of any application. This chapter describes those alternatives and the criteria you should use to select which one is right for you. For a general description of templates, see the online *Language Reference*.

CAUTION:
Do not attempt to link objects produced from compiling the assembler listings of programs containing templates. Even if the listing does compile, it will not link correctly. (The linker may perform the link without error but the `.EXE` will produce incorrect results.)

Template Terms

The following terms are used to describe the template constructs in C++:

class template
A template used to generate classes. Classes generated in this fashion are called `template classes`. A class template describes a family of related classes. It can simply be a declaration, or it can can be a definition of a particular class.

function template
A template used to generate functions. Functions generated in this fashion are called `template functions`. A function template describes a family of related functions. It can simply be a declaration, or it can be a definition of a particular function.

declaring template
A class template or function template that includes a declaration but does not include a definition. For example, this is what a declaring function template would look like.

```
template<class A> void foo(A*a);
```

A declaring class template would look like this

```
template<class T> class C;
```

defining template
A class template or function template declaration that includes a definition. A defining function template would look like this:

```
template<class A> void foo(A*a) {a ->Bar();};
```

A defining class template would look like this:

```
template<class T> class C : public A {public: void boo();};
```

explicit definition
A user-supplied definition that overrides a template. For example, an explicit definition of the `foo()` function would look like this:

```
void foo(int *a) {a++;}
```

An explicit definition of a template class looks like this:

```
class C<short> {
  public: int moo();
 }
```

Instantiation
A defining template defines a whole family of classes or functions. An *instantiation* of a template class or function is a a specific class or function that is based on the template.

How the Compiler Expands Templates

You can choose from three alternatives for instantiating templates:

1. Including defining templates everywhere. See “Including Defining Templates Everywhere” on page 137 for more details.
2. Using VisualAge C++'s automatic facility to ensure that there is a single instantiation of the template. See “Structuring for Automatic Instantiation” on page 137 for more details.
3. Manually structuring your code so that there is a single instantiation of the template. See “Manually Structuring for Single Instantiation” on page 142 for more details.

If you want to make the best choice amongst these alternatives, it is easiest if you first understand how the compiler reacts when it encounters templates. When you use templates in your program, the VisualAge C++ compiler automatically instantiates each defining template that is:

- Referenced in the source code

- Visible to the compiler (included as the result of an `#include` statement)
- Not explicitly defined by the programmer

If an application consists of several separate compilation units that are compiled separately, it is possible that a given template is expanded in two or more of the compilation units. For templates that define classes, inline functions, or static nonmember functions, this is usually the desired behaviour. These templates normally need to be defined in each compilation unit where they are used.

For other functions and for static data members, which have external linkage, defining them in more than one compilation unit would normally cause an error when the program is linked. VisualAge C++ avoids this problem by giving special treatment to template-generated versions of these objects. At link time, VisualAge C++ gathers all template-generated functions and static member definitions, plus any explicit definitions, and resolves references to them in the following manner:

- If an explicit definition of the function or static member exists, it is used for all references. All template-generated definitions of that function or static member are discarded.
- If no explicit definition exists, one of the template-generated definitions is used for all references. Any other template-generated definitions of that function or static member are discarded.

Note that you may have only one explicit definition of any external linkage template instance.

Example of Generating Template Function Definitions

The class template `Stack` provides an example of template function generation. `Stack` implements a stack of items. The overloaded operators << and >> are used to push items on to the stack and pop items from the stack. Assume that the declaration of the `Stack` class template is contained in the file `stack.h`:

```
template <class Item, int size> class Stack {
  public:
      int operator << (Item item);  // push operator
      int operator >> (Item& item); // pop operator
      Stack() { top = 0; }          // constructor defined inline
   private:
      Item stack[size];             // stack of items
      int   top;                    // index to top of stack
};
```

Figure 13. Declaration of Stack in stack.h

In the template, the constructor function is defined inline. Assume the other functions are defined using separate function templates in the file stack.c:

```
template <class Item, int size>
   int Stack<Item,size>::operator << (Item item) {
     if (top >= size) return 0;
      stack[top++] = item;
      return 1;
   }
template <class Item, int size>
    int Stack<Item,size>::operator >> (Item& item)
   {
      if (top <= 0) return 0;
       item = stack[--top];
       return 1;
   }
```

Figure 14. Definition of operator functions in stack.c

In this example, the constructor has internal linkage because it is defined inline in the class template declaration. In each compilation unit that uses an instance of the Stack class, the compiler generates the constructor function body. Each unit uses its own copy of the constructor.

In each compilation unit that includes the file stack.c, for any instance of the Stack class in that unit, the compiler generates definitions for the following functions (assuming there is no explicit definition) :

```
Stack<item,size>::operator<<(item)
Stack<item,size>::operator>>(item&)
```

For example, given the following source file stack.cpp:

```
#include "stack.h"
#include "stack.c"

void Swap(int i&, Stack<int,20>& s)
{
   int j;
   s >> j;
   s << i;
   i = j;
}
```

the compiler generates the functions Stack<int,20>::operator<<(int) and Stack<int,20>::operator>>(int&) because both those functions are used in the program, their defining templates are visible, and no explicit definitions were seen.

Including Defining Templates

The following sections describe the three methods of including defining templates and how they would be applied to this example. The methods are:

- including defining templates everywhere
- structuring for automatic instantiation
- manually stucturing for single instantiation

Automatic instantiation is the recommended method.

Including Defining Templates Everywhere

The simplest way to instantiate templates is to include the defining template in every compilation unit that uses the template. This alternative has the following disadvantages:

- If you make even a trivial change to the implementation of a template, you must recompile every compilation unit that uses it.
- The compilation process is slower, and the resulting object files are bigger because the templates are expanded in every compilation unit where they are used. Note, however, that the duplicated code for the templates is eliminated during linking, so the executable files are not larger if you choose to include defining templates everywhere.

For example, to use this method with the `Stack` template, include both `stack.h` and `stack.c` in all compilation units that use an instance of the `Stack` class. The compiler then generates definitions for each template function. Each template function may be defined multiple times, increasing the size of the object file.

Structuring for Automatic Instantiation

The recommended way to instantiate templates is to structure your program for their automatic instantiation. The advantages of this method are:

- It is easy to do.
- Unlike the method of including defining templates everywhere, you do not get larger object files and slower compile times.
- Unlike the method of including defining templates everywhere, you do not have to recompile all of the compilation units that use a template if that template implementation is changed.

The disadvantages of this method are:

- It may not be practical in a team programming environment because the compiler may update source files that are being modified at the same time by somebody else.

Structuring for Automatic Instantiation

- The modifications that are made to source files may not be file system independent. For example, header files that are locally available may be included rather than header files that are available on a network.
- There are some situations where the compiler cannot determine exactly which header files should be included.

To use this facility:

1. Declare your template functions in a header file using class or function templates, but **do not define them**. Include the header file in your source code.
2. For each header file, create a *template-implementation* file with the same name as the header and the extension `.c`. Define the template functions in this template-implementation file.

Note: Use the same compiler options to link your object files that you use to compile them. For example, if you compile with the command:

```
icc /C /Gm /Sa myfile.cpp
```

link with the command:

```
icc /Tdp /Gm /Sa myfile.obj
```

This is especially important for options that control libraries, linkage, and code compatibility. This does not apply to options that affect only the creation of object code (for example, `/C` and `/Fo`). (Note that in the compile step, the `/Tdp` was implicit because it is the default for files with the extension `.cpp`.)

For each header file with template functions that need to be defined, the compiler generates a *template-include* file. The template-include file generates `#include` statements in that file for:

- The header file with the template declaration
- The corresponding template-implementation file
- Any other header files that declare types used in template parameters.

Important: If you have other declarations that are used inside templates but are not template parameters, you must place or `#include` them in either the template-implementation file or one of the header files that will be included as a result of the above three steps. Define any classes that are used in template arguments and that are required to generate the template function in the header file. If the class definitions require other header files, include them with the **#include** directive. The class definitions are then available in the template-implementation file when the function definition is compiled. Do not put the definitions of any classes used in template arguments in your source code.

```
foo.h
  template<class T> void foo(T*);

hoo.h
  void hoo(A*);

foo.c
  template<class T> void foo(T* t)
     {t -> goo(); hoo(t);}

other.h
  class A {public: void goo() {} };

main.cpp
  #include "foo.h"
  #include "other.h"
  #include "hoo.h"
  int main() { A a;  foo(&a); }
```

This requires the expansion of the foo(T*) template with "class A" as the template type parameter. The compiler will create a template-include file TEMPINC\foo.cpp. The file contents (simplified below) would be:

```
#include "foo.h"          //the template declaration header
#include "other.h"        //file defining template type parameter
#include "foo.c"          //corresponding template implementation
void foo(A*);             //triggers template instantiation
```

This won't compile properly because the header "hoo.h" didn't satisfy the conditions for inclusion but the header is required to compile the body of foo(A*). One solution is to move the declaration of hoo(A*) into the "other.h" header.

The function definitions in your template-implementation file can be explicit definitions, template definitions, or both. Any explicit definitions are used instead of the definitions generated by the template.

Before it invokes the linker, the compiler compiles the template-include files and generates the necessary template function definitions. Only one definition is generated for each template function.

By default, the compiler stores the template-include files in the TEMPINC subdirectory under the current directory. The compiler creates the `TEMPINC` directory if it does not already exist. To redirect the template-include files to another directory, use the `/Ft`*`dir`* compiler option, where *`dir`* is the directory to contain the template-include files. You can specify a fully-qualified path name or a path name relative to the current directory.

If you specify a different directory for your template-include files, make sure that you specify it consistently for all compilations of your program, including the link step.

Note: After the compiler creates a template-include file, it may add information to the file as each compilation unit is compiled. However, the compiler never removes information from the file. If you remove function instantiations or reorganize your program so that the template-include files become obsolete, you may want to delete one or more of these files and recompile your program. In addition, if error messages are generated for a file in the `TEMPINC` directory, you must either correct the errors manually or delete the file and recompile. To regenerate all of the template-include files, delete the `TEMPINC` directory, the `.OBJ` files, and recompile your program.

If you do not delete the `.OBJ` files, typical MAKEFILE rules will prevent the OBJs from being recompiled, and therefore the template-include files will not be updated with all the lines needed for all the compilation units used in the program. The end result would be that the link would fail.

Example of a Template-Implementation File

In the `Stack` example, the file `stack.c` is a template-implementation file. To create a program using the `Stack` class template, `stack.h` and `stack.c` must reside in the same directory. You would include `stack.h` in any source files that use an instance of the class. The `stack.c` file does not need to be included in any source files. Then, given the source file:

```
#include "stack.h"

void Swap(int i&, Stack<int,20>& s)
{
   int j;
   s >> j;
   s << i;
   i = j;
}
```

the compiler automatically generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)`.

You can change the name of the template-implementation file or place it in a different directory using the `#pragma implementation` directive. This **#pragma** directive has the format:

```
#pragma implementation(path)
```

where *path* is the path name for the template-implementation file. If it is only a partial path name, it must be relative to the directory containing the header file.

Note: This path is a quoted string following the normal conventions for writing string literals. In particular, backslashes must be doubled.

For example, in the `Stack` class, to use the file `stack.def` as the template-implementation file instead of `stack.c`, add the line:

```
#pragma implementation("stack.def")
```

anywhere in the `stack.h` file. The compiler then looks for the template-implementation file `stack.def` in the same directory as `stack.h`.

Example of a Template-Include File

The following example shows the information you would find in a typical template-include file generated by the compiler:

```
/*0000000000*/ #pragma sourcedir("c:\swearsee\src")    0
/*0698421265*/ #include "c:\swearsee\src\list.h"        1
/*0000000000*/ #include "c:\swearsee\src\list.c"        2
/*0698414046*/ #include "c:\swearsee\src\mytype.h"      3
/*0698414046*/ #include "c:\IBMCPP\INCLUDE\iostream.h"  4
#pragma define(List<MyType>)                           5
ostream& operator<<(ostream&,List<MyType>);            6
#pragma undeclared                                     7
int count(List<MyType>);                               8
```

0 This pragma ensures that the compiler will look for nested include files in the directory containing the original source file, as required by the VisualAge C++ file inclusion rules.

1 The header file that corresponds to the template-include file. The number in comments at the start of each **#include** line (for this line `/*0698421265*/`) is a time stamp for the included file. The compiler uses this number to determine if the template-include file is current or should be recompiled. A time stamp containing only zeroes (`0`) as in line **2** means the compiler is to ignore the time stamp.

2 The template-implementation file that corresponds to the header file in line **1**

3 Another header file that the compiler requires to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point.

4 Another header file required by the compiler. It is referenced in the function declaration in line **6**.

5 The compiler inserts `#pragma define` directives to force the definition of template classes. In this case, the class `List<MyType>` is to be defined and its member functions are to be generated.

6 The `operator<<` function is a nonmember function that matched a template declaration in the `list.h` file. The compiler inserts this declaration to force the generation of the function definition.

7 The `#pragma undeclared` directive is used only by the compiler and only in template-include files. All template functions that were explicitly declared in at least one compilation unit appear before this line. All template functions that were called, but never declared, appear after this line. This division is necessary because the C++ rules for function overload resolution treat declared and undeclared template functions differently.

8 `count` is an example of a template function that was called but not declared. The template declaration of the function must have been contained in `list.h`, but the instance `count(List<MyType>)` was never declared.

Note: Although you can edit the template-include files, it is not normally necessary or advisable to do so.

Manually Structuring for Single Instantiation

If you do not want to use the automatic instantiation method of generating template function definitions, you can structure your program in such a way that you define template functions directly in your compilation units. The advantages of this approach are:

- Object files are smaller and compile times are shorter than they are when you include defining templates everywhere. When you structure your code manually for template instantiation, you avoid the potential problems that automatic instantiation can cause, such as dependency on a particular file system or file sharing problems.

There are also disadvantages to structuring your code manually for template instantiation:

- You have to do more work than for the other two methods. You may have to reorganize source files and create new compilation units.
- You have to be aware of all of the instantiations of templates that are required by the entire program.

Manually Structuring for Single Instantiation

Note: It is recommended that you use the compiler's automatic instantiation facility. The manual structuring method described here is useful if you find you cannot work around the limitations of the automatic instantiation method.

Use `#pragma define` directives to force the compiler to generate the necessary definitions for all template classes used in other compilation units. Use explicit declarations of non-member template functions to force the compiler to generate them.

To use the second method, include `stack.h` in all compilation units that use an instance of the `Stack` class, but include `stack.c` in only one of the files. Alternatively, if you know what instances of the `Stack` class are being used in your program, you can define all of the instances in a single compilation unit. For example:

```
#include "stack.h"
#include "stack.c"
#include "myclass.h"  // Definition of "myClass" class
#pragma define(Stack<int,20>)
#pragma define(Stack<myClass,100>)
```

The `#pragma define` directive forces the definition of two instances of the `Stack` class without creating any object of the class. Because these instances reference the member functions of that class, template function definitions are generated for those functions. See the online *Language Reference* for more information about the **#pragma** directive.

You can compile and link in one step or two, but you must use `icc` to invoke the linker. For example, to compile and link `stack.cpp`, you could use the command:

```
icc stack.cpp
```

or the commands:

```
icc /C stack.cpp
icc /Tdp stack.obj
```

(Note that in the first example, you need not specify the `/Tdp` option because it is the default for files with the extension `.cpp`.)

When you use these methods, you may also need to specify the `/Ft-` option to ensure that the compiler does not also automatically create the `TEMPINC` files according to the automatic generation facility.

Mixing Old and New Templates

The way that VisualAge C++ resolves template functions and data is new for this release of VisualAge C++. Previous releases used a different method (called the *old method* in this section). The current method is an improvement because it gives faster link times and reduces the size of the executable or DLL that is output.

In previous releases of VisualAge C++, when multiple template functions were resolved to a single definition, the unused template generated functions were not removed and became “dead code” in the linker output. The current release of VisualAge C++ does not do this. Because of this, the option of including defining templates everywhere (see “Including Defining Templates Everywhere” on page 137 for details) no longer results in larger executables.

If you want to link together objects containing template functions generated by the current release of VisualAge C++ and objects containing template functions generated by previous releases, you must link with `ICC` using the `/Gk+` option.

When you specify this option, `ICC` uses the template resolution method from previous releases of VisualAge C++. (In addition, this option causes `ICC` to supply the `/OLDCPP` option to the linker.) When you link objects containing template functions using `ICC` with the `/Gk+` option:

- If a user-supplied template function exists, it will be selected. If a template-generated function body was created by the old method, it will not be removed from the executable, but if it was created by the current method it will be removed.
- If no user-supplied function exists, but one or more template-generated functions created by the old method does exist, then one of these template-generated functions will be selected. The other template-generated functions created by the old method will not be removed, but those created by the current method will be removed.
- If no user-supplied function and no template-generated functions created by the old method exist, one new template-generated function will be selected. Other template-generated functions created by the current method will be removed.

Note: If you do not specify the `/Gk+` option, the linker produces an error message asking you to link using `/OLDCPP`. If you get this error message, you should relink using `ICC` with the `/Gk+` option.

The current method of resolving template static data members is also better than the method used in previous releases of VisualAge C++. The old method had the following limitations:

- You could not combine template definitions of static data members with explicit definitions. If you tried to use a static member template in one compilation unit and an explicit definition in another, the linker generated an error about multiple definitions.
- Static data members defined by templates were not visible as dictionary entries in libraries. If your program referenced a static member defined in a library object, but did not reference any other external symbols in that object, the linker would not extract the object from the library.

When you use the current release of VisualAge C++ to create the objects that are linked together, these restrictions no longer apply. If you mix object files that were created using a previous release with object files created using the current release, these limitations still apply to static members for which one or more old template-generated instances exists in the old objects.

Mixing Old and New Templates

Calling Conventions

This chapter describes the calling conventions used by the VisualAge C++ compiler for both C and C++:

- **_Optlink**
- **_System**
- **_Pascal** and **_Far32 _Pascal**
- **__stdcall**
- **__cdecl**
- 32/16-bit conventions:
  - **_Far16_Cdecl**
  - **_Far16 _Pascal**
  - **_Far16 _Fastcall**

The **_Optlink** convention is specific to VisualAge C++ compiler and is the fastest method of calling C or C++ functions or assembler routines, but it is not standard for all OS/2 applications. The **_Optlink** calling convention is described in more detail in "**_Optlink** Calling Convention" on page 150.

The **_System** calling convention, while slower, is standard for all OS/2 applications and is used for calling OS/2 APIs. For a description of the **_System** calling convention, see "**_System** Calling Convention" on page 170.

The **_Pascal** and **_Far32 _Pascal** conventions are used to develop virtual device drivers. The **_Far32 _Pascal** convention can only be used for applications written in C that run at ring zero (compiled with the `/Gr+` option). More information about the **_Pascal** and **_Far32 _Pascal** conventions can be found in "_Pascal and _Far32_Pascal Calling Conventions" on page 178.

Notes:

1. You cannot call a function using a different calling convention than the one with which it is compiled. For example, if a function is compiled with **_System** linkage, you cannot later call it specifying **_Optlink** linkage.
2. VisualAge C++ does not allow functions that use the **__stdcall** calling convention to have both the following characteristics:
   - No prototype
   - A variable number of arguments.

   In particular, an unprototyped function that accepts a variable number of arguments and uses the **__stdcall** calling convention will not link. This is because **__stdcall** functions are referenced in `.OBJ` files using a name that is a combination of the function name and the number of parameters that the function takes. Therefore, if one compilation has defined it with a different number of parameters than another compilation unit, the two references to the function will have different external function names. The linker will not be able to resolve them.

With prototyping, the compiler encodes the name in such a way that the number of parameters becomes extraneous (irrelevant) information and so the two references end up with the same external function name.

The different methods of calling 16-bit code from the VisualAge C++ compiler and the 16-bit calling conventions are discussed in Chapter 12, "Calling between 32-Bit and 16-Bit Code" on page 191.

You can specify any of the calling conventions for all functions within a program using the `/Mp` (for **_Optlink**) or `/Ms` (for **_System**) compiler option. You can also use the `/Mt` option to specify **__stdcall** and the `/Mc` option to specify **__cdecl**.

Using Linkage Keywords to Specify the Calling Convention

In addition to using options to specify the calling convention for all of the functions in a program, you can also use linkage keywords to specify the calling convention for individual functions. The linkage keywords and their equivalent calling convention suboptions of `/Mp` and `/Ms` are listed in the following table. Linkage keywords take precedence over the compiler option, if both are specified.

Figure 15 (Page 1 of 2). Equivalent Linkage Keywords and Compiler Suboptions

Linkage Keyword	**Equivalent Compiler Suboption**
extern "SYSTEM"	**_System**
extern "OPTLINK"	**_Optlink**
extern "FAR16 CDECL"	**_Far16_Cdecl**
extern "FAR16 PASCAL"	**_Far16 _Pascal**
extern "FAR16 FASTCALL"	**_Far16 _Fastcall**
extern "C"	**_Optlink** suboption for /Mp, `system` suboption for /Ms
extern "C++"	**_Optlink** with mangling, overloading, etc.
extern "BUILTIN"	**`_Builtin`**
extern "PASCAL"	**_Pascal**
extern "**__stdcall**"	**__stdcall**

Figure 15 (Page 2 of 2). Equivalent Linkage Keywords and Compiler Suboptions

Linkage Keyword	Equivalent Compiler Suboption
extern "**__cdecl**"	**__cdecl**

Note: C++ member functions always use the **_Optlink** calling convention, but they also have long internal names (called *mangled* names) that encode their containing class name and parameter types. Nonstatic member functions also have an implicit `this` parameter that refers to the object on which they are invoked. This combination of **_Optlink** rules, name mangling, and the `this` parameter is called *C++ linkage*, and it is considered distinct from **_Optlink** alone.

In C programs, you can also use **`#pragma linkage`** for all but **__stdcall** and **__cdecl**. However, you should using the pragma and use the linkage keywords discussed above instead. Problems encountered when using **`#pragma linkage`** include:

- it does not exist in C++, so it can't be used in any headers that will be used in C and C++.
- it is difficult to mark function pointers with **`#pragma linkage`**, especially if they are in structures or hidden behind arrays.
- code which uses **`#pragma linkage`** tends to be less clear than code using the other methods described earlier.

The following examples illustrate the drawbacks.

```
/* Defining a callback function using #pragma linkage */
typedef int foo(void);
#pragma linkage(foo, system)
struct ss {
  int x:
  foo callback;
};
```

Observe the improved clarity when the same code is written using keywords:

```
/* Defining a callback function using keywords */
struct ss {
  int x;
  int (* _System callback)(void);
};
```

Furthermore, there are cases where necessary conversions don't happen when **`#pragma linkage`** is used. Consider the following:

```
void func16(char *);           /*Parameter needs to be converted */
func() {
  func16(pointer_variable);      /* Conversion doesn't happen... */
}
#pragma linkage (func16, far16 pascal) /* ... because pragma linkage
                                          occurs after the call */
```

Here's the equivalent code using keywords:

```
void _Far16 _pascal func16(char *);
func() {
  func16(pointer_variable):     /* Conversion does happen properly */
}
```

See the *User's Guide* for more details on setting the calling convention and on compiler options. For information about linkage keywords and **#pragma linkage**, see the online *Language Reference*.

_Optlink Calling Convention

This is the default calling convention. It is an alternative to the **_System** convention that is normally used for calls to the operating system. It provides a faster call than the **_System** convention, while ensuring conformance to the ANSI and SAA language standards.

You can explicitly give a function the **_Optlink** convention with the **_Optlink** keyword.

Features of _Optlink

- Parameters are pushed from right to left onto the stack to allow for varying length parameter lists without having to use hidden parameters.
- The caller cleans up the parameters.
- The general-purpose registers EBP, EBX, EDI, and ESI are preserved across the call.
- The general-purpose registers EAX, EDX, and ECX are not preserved across the call.
- Floating-point registers are not preserved across the call.
- The three conforming parameters that are lexically leftmost (conforming parameters are 1, 2, and 4-byte signed and unsigned integers, enumerations, and all pointer types) are passed in EAX, EDX, and ECX, respectively.
- Up to four floating-point parameters (the lexically first four) are passed in extended-precision format (80-bit) in the floating-point register stack.
- All other parameters are passed on the 80386 stack.
- Space for the parameters in registers is allocated on the stack, but the parameters are not copied into that space.
- Conforming return values are returned in EAX.

- Floating-point return values are returned in extended-precision format in the topmost register of the floating-point stack.
- When you call external functions, the floating-point register stack contains only valid parameter registers on entry and valid return values on exit. (When you call functions in the current compilation unit that do not call any other functions, this state may not be true.)
- Under some circumstances, the compiler will not use EBP to access automatic and parameter values, thus increasing the efficiency of the application. Whether it is used or not, EBP will not change across the call.
- Calls with aggregates returned by value pass a hidden first parameter that is the address of a storage area determined by the caller. This area becomes the returned aggregate. The hidden pointer parameter is always considered "nonconforming", and is not passed in a register. The called function must load it into EAX before returning.
- The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function and undefined on exit.
- The compiler will not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.
- In a prototyped function taking a variable number of parameters (that is, one whose parameter list ends in an elipsis), only the parameters preceding the elipsis are eligible to be passed in registers.

Tips for Using _Optlink

To obtain the best performance when using the **_Optlink** convention, follow these tips:

- Prototype all function declarations for better performance. The C++ language requires all functions to have prototypes.

  Note: All calls and functions must be prototyped consistently; that is, functions declared more than once must have identical prototypes. If prototyping is not consistent, the results will be undefined.
- Place the conforming and floating-point parameters that are most heavily used lexically leftmost in the parameter list so they will be considered for registers first. If they are adjacent to each other, the preparation of the parameter list will be faster.
- If you have a parameter that is only used near the end of a function, put it at or near the end of the parameter list. If all of your parameters are only used near the end of functions, consider using **_System** linkage.

- If you are passing structures by value, put them at the end of the parameter list.
- Avoid using variable arguments in nonprototype functions. This practice results in undefined behavior under the ANSI C standard.
- If you have a variable-length argument list, consider using **_System** linkage. It is faster in this situation.
- Compile with optimization on by specifying `/O+`.

For additional tips on how to improve the performance of your program, see Chapter 4, "Optimizing Your Program" on page 35.

General-Purpose Register Implications

EAX, EDX, and ECX are used for the lexically first three conforming parameters with EAX containing the first parameter, EDX the second, and ECX the third. Four bytes of stack storage are allocated for each register parameter that is present, but the parameters exist only in the registers at the time of the call.

If there is no prototype or an incomplete prototype for the function called, an eyecatcher is placed after the call instruction to tell the callee how the register parameters correspond to the stack storage mapped for them. Fully prototyped code never needs eyecatchers.

Eyecatchers

An eyecatcher is a recognizable sequence of bytes that tells unprototyped code which parameters are passed in which registers. Eyecatchers apply only to code without prototype statements.

The eyecatcher instruction is placed after the call instruction for a nonprototype function. The choice of instruction for the eyecatcher relies on the fact that the TEST instruction does not modify the referenced register, meaning that the return register of the call instruction is not modified by the eyecatcher instruction. The absence of an eyecatcher in unprototyped code implies that there are no parameters in registers. (Note that this eyecatcher scheme does not allow the use of execute-only code segments.)

The eyecatcher has the format:

```
TEST EAX, immed32
```

Note that the short-form binary encoding (A9) of `TEST EAX` must be used for the eyecatcher instruction.

The 32-bit immediate operand is interpreted as a succession of 2-bit fields, each of which describes a register parameter or a 4-byte slot of stack memory. Because only one 32-bit read of the eyecatcher is made, only 24 bits of the immediate operand are loaded. The actual number of parameters that can be considered for registers is restricted to 12.

Because of byte reversal, the bits that are loaded are the low-order 24 bits of the 32-bit immediate operand. The highest-order 2-bit field of the 24 bits analyzed corresponds to the lexically first parameter, while subsequent parameters correspond to the subsequent lower-order 2-bit fields. The meaning of the values of the fields is as follows:

Value	**Meaning**
00	This value indicates that there are no parameters remaining to be put into registers, or that there are parameters that could be put into registers but there are no registers remaining. It also indicates the end of the eyecatcher.
01	The corresponding parameter is in a general-purpose register. The leftmost field of this value has its parameter in EAX, the second leftmost (if it exists) in EDX, and the third (if it exists) in ECX.
10	The corresponding parameter is in a floating-point register and has 8 bytes of stack reserved for it (that is, it is a `double`). ST(0), ST(1), ST(2), and ST(3) contain the lexically-first four floating-point parameters (fewer registers are used if there are fewer floating-point parameters). ST(0) contains the lexically first (leftmost 2-bit field of type 10 or 11) parameter, ST(1) the lexically second parameter, and so on.
11	The corresponding parameter is in a floating-point register and has 16 bytes of stack reserved for it (that is, it is a `long double`).

Examples of Passing Parameters

The examples on the following pages are included for purposes of illustration and clarity only. They have not been optimized. These examples assume that you are familiar with programming in assembler. Note that, in each example, the stack grows toward the bottom of the page, and `ESP` always points to the top of the stack.

Examples Using _Optlink

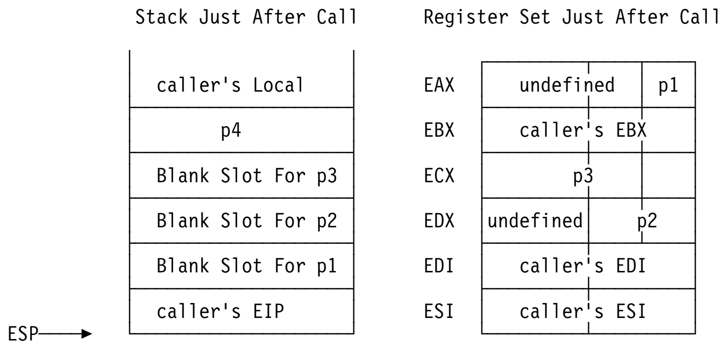
Passing Conforming Parameters to a Prototyped Routine

The following example shows the code sequences and a picture of the stack for a call to the function `foo`:

```
long foo(char p1, short p2, long p3, long p4);

short x;
long y;

y = foo('A', x, y+x, y);
```

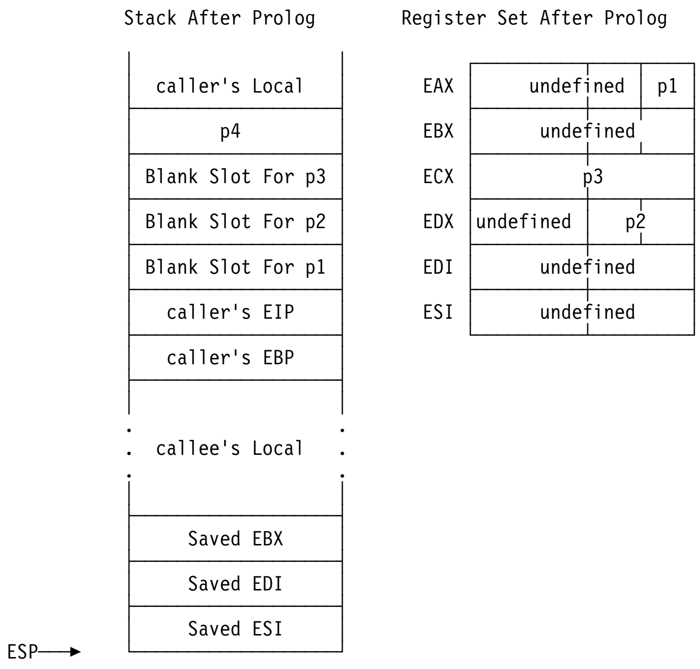
Caller's code surrounding call:

```
PUSH     y           ; Push p4 onto the 80386 stack
SUB      ESP, 12     ; Allocate stack space for
                     ;  register parameters
MOV      AL, 'A'     ; Put p1 into AL
MOV      DX,  x      ; Put p2 into DX
MOVSX    ECX, DX     ; Sign-extend x to long
ADD      ECX, y      ; Calculate p3 and put it into ECX
CALL     FOO         ; Make call
```



Callee's prolog code:

```
PUSH     EBP                        ; Save caller's EBP
MOV      EBP, ESP                   ; Set up callee's EBP
SUB      ESP, callee's local size ; Allocate callee's Local
PUSH     EBX                        ; Save preserved registers -
PUSH     EDI                        ;  will optimize to save
PUSH     ESI                        ;  only registers callee uses
```

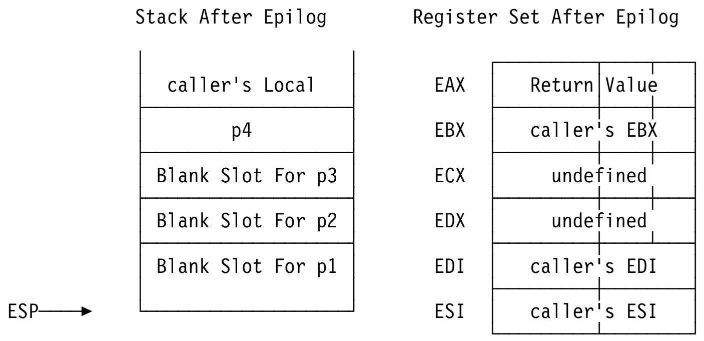


Note: The term `undefined` in registers EBX, EDI, and ESI refers to the fact that they can be safely overwritten by the code in `foo`.

Callee's epilog code:

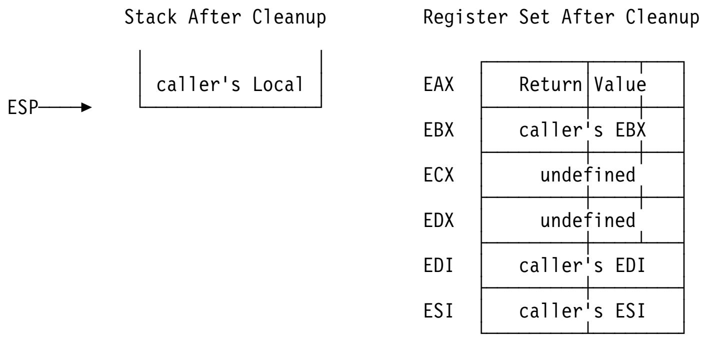
```
MOV      EAX, RetVal ; Put return value in EAX
POP      ESI         ; Restore preserved registers
POP      EDI
POP      EBX
MOV      ESP, EBP    ; Deallocate callee's local
POP      EBP         ; Restore caller's EBP
RET                  ; Return to caller
```

Examples Using _Optlink



Caller's code just after call:

```
ADD      ESP, 16      ; Remove parameters from stack
MOV      y,   EAX     ; Use return value.
```



Passing Conforming Parameters to an Unprototyped Routine

This example differs from the previous one by providing:

- An eyecatcher after the call to `foo` in the caller's code
- The code necessary to perform the default widening rules required by ANSI
- The instruction to clean up the parameters from the stack.

If `foo` were an ellipsis routine with fewer than three conforming parameters in the invariant portion of its parameter list, it would also include the code to interpret the eyecatchers in its prolog.

```
y = foo('A', x, y+x, y);
```

Caller's code surrounding call:

```
PUSH     y               ; Push p4 onto the 80386 stack
SUB      ESP, 12         ; Allocate stack space for register parameters
MOV      EAX, 00000041h  ; Put p1 into EAX (41 hex = A ASCII)
MOVSX    EDX, x          ; Put p2 into EDX
MOV      ECX, y          ; Load y to calculate p3
ADD      ECX, x          ; Calculate p3 and put it into ECX
CALL     FOO             ; Make call
TEST     EAX, 00540000h  ; Eyecatcher indicating 3 general-purpose
                         ;  register parameters (see page 152)
ADD      ESP, 16         ; Clean up parameters after return
```

Passing Floating-Point Parameters to a Prototyped Routine

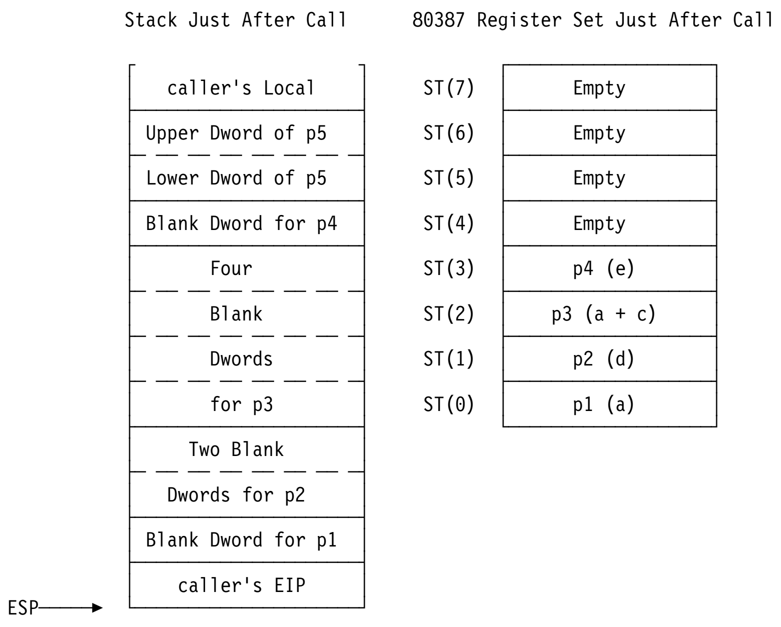
The following example shows code sequences, 80386 stack layouts, and floating-point register stack states for a call to the routine `fred`. For simplicity, the general-purpose registers are not shown.

```
double fred(float p1, double p2, long double p3, float p4, double p5);

double a, b, c;
float  d, e;

a = b + fred(a, d, (long double)(a + c), e, c);
```

Examples Using _Optlink

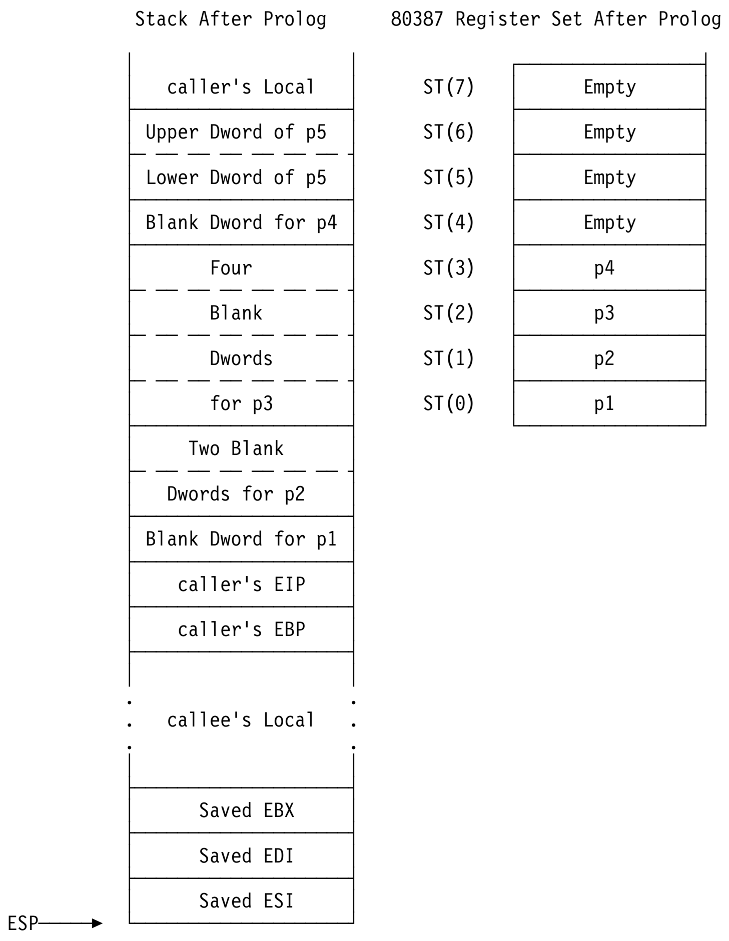
Caller's code up until call:

```
PUSH    2ND DWORD OF c       ; Push upper 4 bytes of c onto stack
PUSH    1ST DWORD OF c       ; Push lower 4 bytes of c onto stack
FLD     DWORD_PTR e          ; Load e into 80387, promotion
                             ;  requires no conversion code
FLD     QWORD_PTR a          ; Load a to calculate p3
FADD    ST(0), QWORD_PTR c   ; Calculate p3, result is long double
                             ;  from nature of 80387 hardware
FLD     QWORD_PTR d          ; Load d, no conversion necessary
FLD     QWORD_PTR a          ; Load a, demotion requires conversion
FSTP    DWORD_PTR [EBP - T1] ; Store to a temp (T1) to convert to float
FLD     DWORD_PTR [EBP - T1] ; Load converted value from temp (T1)
SUB     ESP, 32              ; Allocate the stack space for
                             ;  parameter list
CALL    FRED                 ; Make call
```



Callee's prolog code:

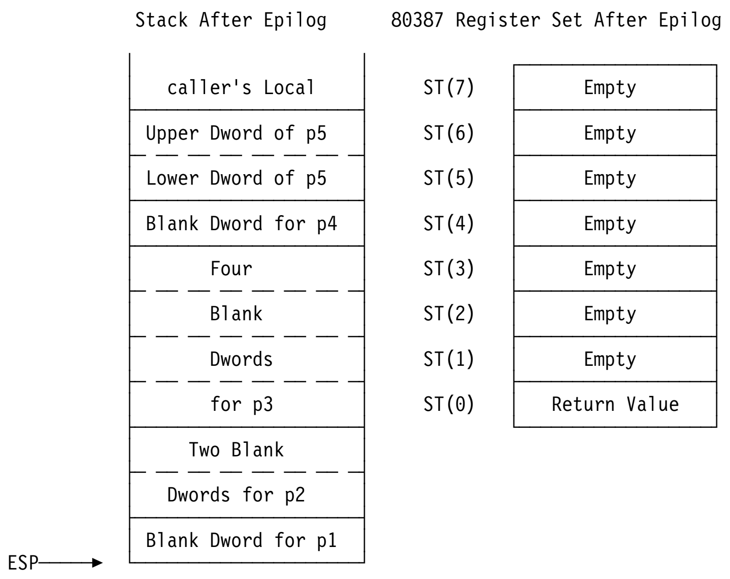
```
PUSH     EBP                       ; Save caller's EBP
MOV      EBP, ESP                  ; Set up callee's EBP
SUB      ESP, callee's local size ; Allocate callee's Local
PUSH     EBX                       ; Save preserved registers -
PUSH     EDI                       ;  will optimize to save
PUSH     ESI                       ;  only registers callee uses
```



Examples Using _Optlink

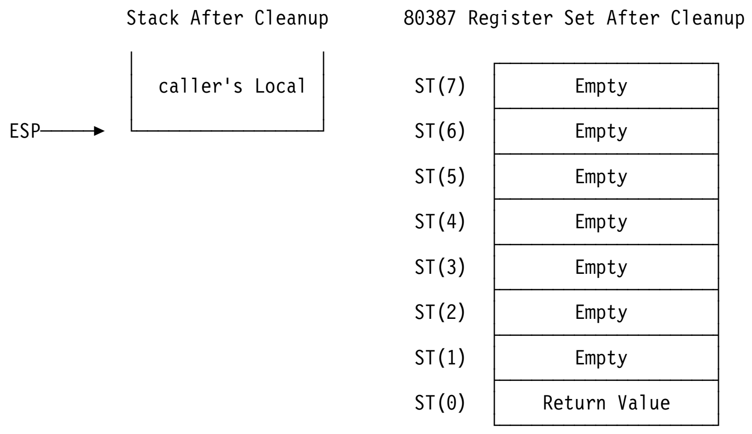
Callee's epilog code:

```
FLD       RETVAL      ; Load return value onto floating-point stack
POP       ESI         ; Restore preserved registers
POP       EDI
POP       EBX
MOV       ESP, EBP    ; Deallocate callee's local
POP       EBP         ; Restore caller's EBP
RET                   ; Return to caller
```



Caller's code just after call:

```
ADD      ESP, 40      ; Remove parameters from stack
FADD     QWORD_PTR b  ; Use return value
FSTP     QWORD_PTR a  ; Store expression to variable a
```



Passing Floating-Point Parameters to an Unprototyped Routine

This example differs from the previous floating-point example by the presence of an eyecatcher after the call to `fred` in the caller's code and the code necessary to perform the default widening rules required by ANSI.

```
double a, b, c;
float  d, e;

a = b + fred(a, d, (long double)(a + c), e, c);
```

Caller's code up until call:

```
PUSH    2ND DWORD OF c       ; Push upper 4 bytes of c onto stack
PUSH    1ST DWORD OF c       ; Push lower 4 bytes of c onto stack
FLD     DWORD_PTR e          ; Load e into 80387, promotion
                             ;  requires no conversion code
FLD     QWORD_PTR a          ; Load a to calculate p3
FADD    ST(0), QWORD_PTR c   ; Calculate p3, result is long double
                             ;  from nature of 80387 hardware
FLD     QWORD_PTR d          ; Load d, no conversion necessary
FLD     QWORD_PTR a          ; Load a, no conversion necessary
SUB     ESP, 40              ; Allocate the stack space for
                             ;  parameter list
CALL    FRED                 ; Make call
TEST    EAX, 00ae0000h       ; Eyecatcher maps the register parameters
ADD     ESP, 48              ; Clean up parameters from stack
```

Examples Using _Optlink

Passing and Returning Aggregates by Value to a Prototyped Routine

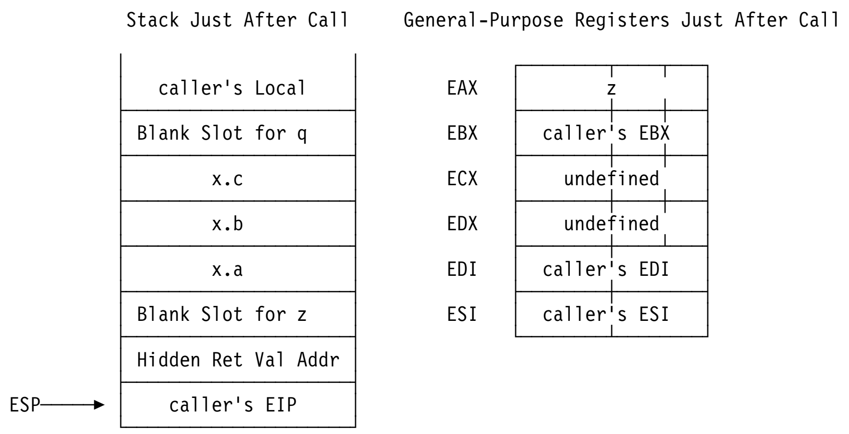
If an aggregate is passed by value, the following code sequences are produced for the caller and callee:

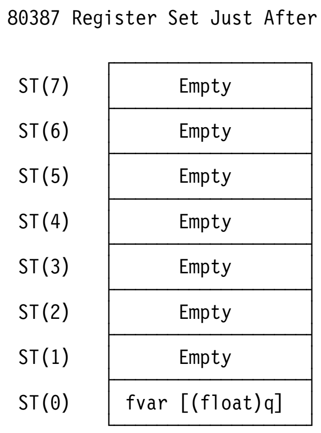
```
'C' Source:

   struct s_tag {
             long  a;
             float b;
             long  c;
             } x, y;
   long  z;
   double q;

   /* Prototype */
   struct s_tag bar(long lvar, struct s_tag aggr, float fvar);

  ⋮

   /* Actual Call */
   y = bar(z, x, q);

  ⋮

   /* callee */
   struct s_tag bar(long lvar, struct s_tag aggr, float fvar)
   {
      struct s_tag temp;

      temp.a = lvar + aggr.a + 23;
      temp.b = fvar - aggr.b;
      temp.c = aggr.c

      return temp;
   }
```

Caller's code up until call:

```
FLD     QWORD_PTR q          ; Load lexically first floating-point
                             ;  parameter to be converted
FSTP    DWORD_PTR [EBP - T1] ; Convert to formal parameter type by
FLD     DWORD_PTR [EBP - T1] ; Storing and loading from a temp (T1)
SUB     ESP, 4               ; Allocate space for the floating-point
                             ;  register parameter
PUSH    x.c                  ; Push nonconforming parameters on
PUSH    x.b                  ;  stack
PUSH    x.a                  ;
MOV     EAX, Z               ; Load lexically first conforming
                             ;  parameter into EAX
SUB     ESP, 4               ; Allocate stack space for the first
                             ;  general-purpose register parameter.
PUSH    addr y               ; Push hidden first parameter (address of
                             ;  return space)
CALL    BAR
```

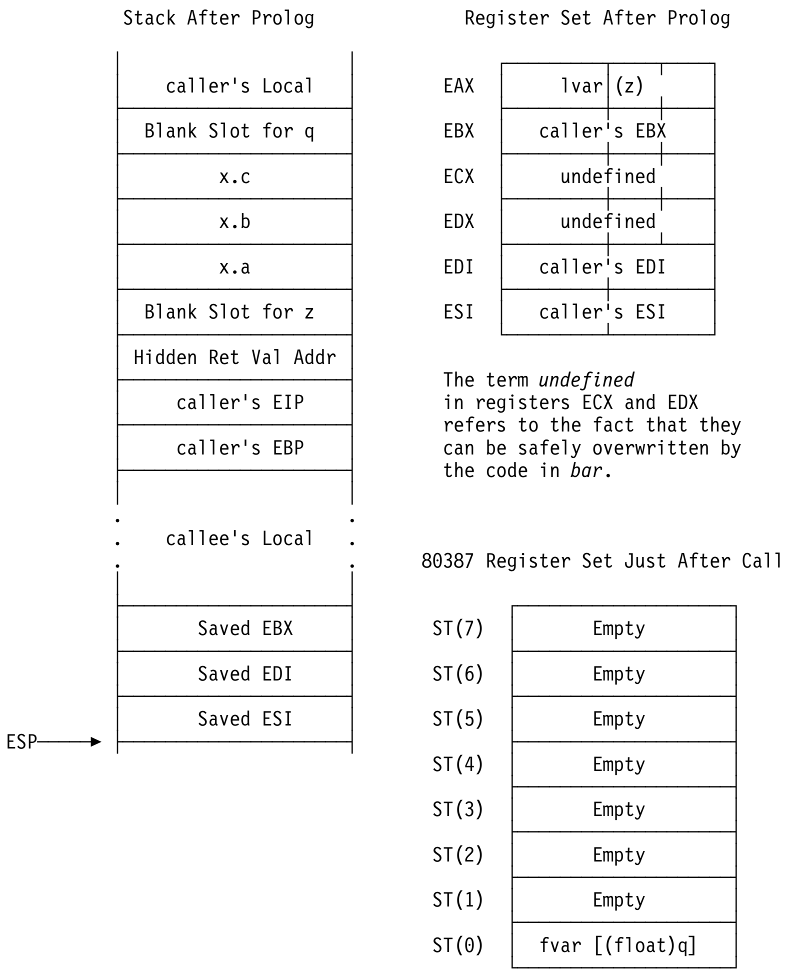
Examples Using _Optlink





Callee's prolog code:

```
PUSH    EBP         ; Save caller's EBP
MOV     EBP, ESP    ; Set up callee's EBP
SUB     ESP, 12     ; Allocate callee's Local
                    ;  = sizeof(struct s_tag)
PUSH    EBX         ; Save preserved registers -
PUSH    EDI         ;  will optimize to save
PUSH    ESI         ;  only registers callee uses
```

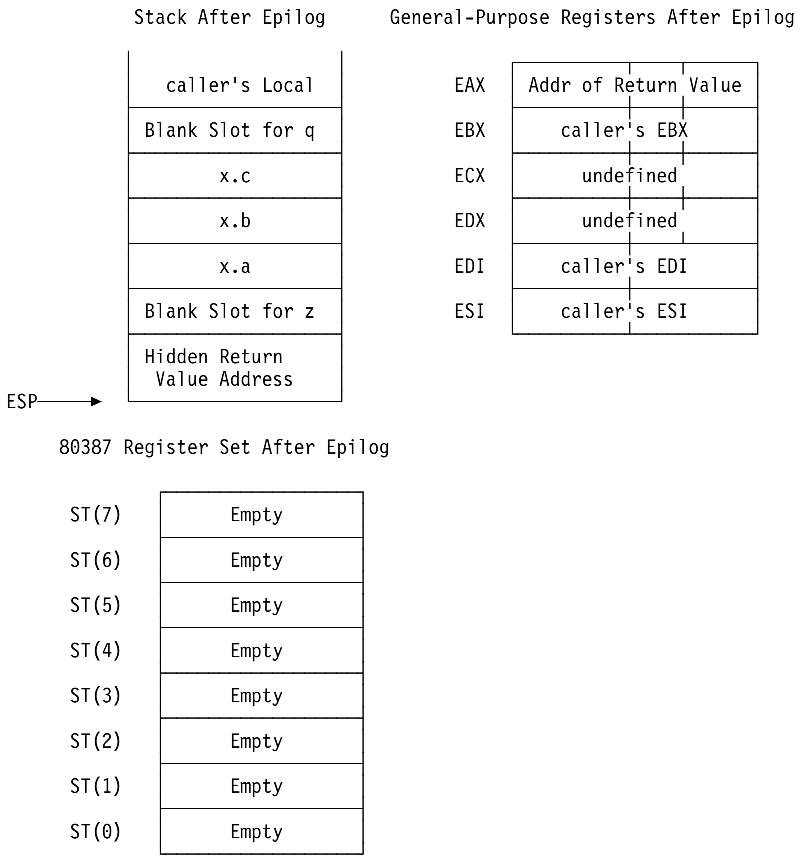


The term *undefined* in registers ECX and EDX refers to the fact that they can be safely overwritten by the code in *bar*.

Examples Using _Optlink

Callee's code:

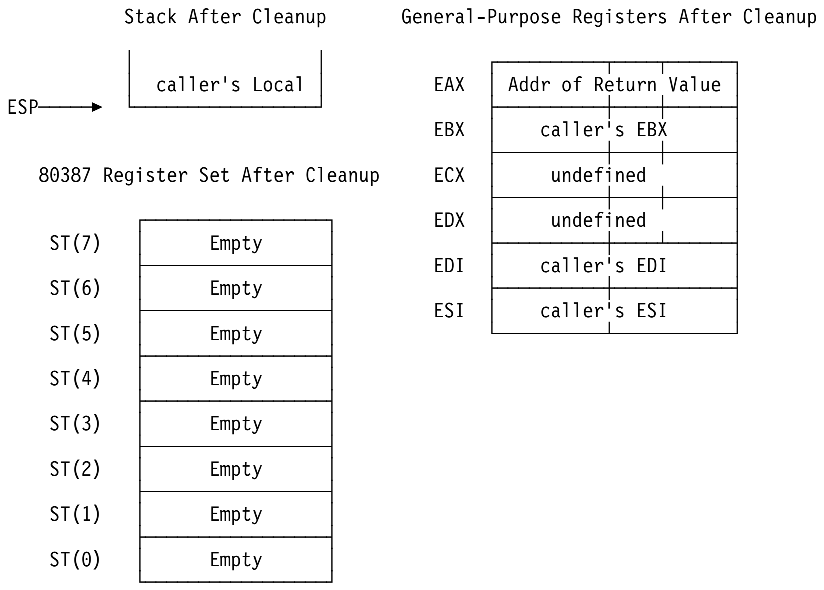
```
temp.a = lvar + aggr.a + 23;
temp.b = fvar - aggr.b;
temp.c = aggr.c

return temp;

ADD       EAX, 23               ;
ADD       EAX, [EBP + 16]       ; Calculate temp.a
MOV       [EBP - 12], EAX       ;

FSUB      DWORD_PTR [EBP + 20]  ; Calculate temp.b
FSTP      DWORD_PTR [EBP - 8]   ;

MOV       EAX, [EBP + 24]       ; Calculate temp.c
MOV       [EBP - 4], EAX        ;

MOV       EAX, [EBP + 8]        ; Load hidden parameter (address
                                ;  of return value storage). Useful
                                ;  both for setting return value
                                ;  and for returning address in EAX.

MOV       EBX, [EBP - 12]       ; Return temp by copying its contents
MOV       [EAX], EBX            ;  to the return value storage
MOV       EBX, [EBP - 8]        ;  addressed by the hidden parameter.
MOV       [EAX + 4], EBX        ;  String move instructions would be
MOV       EBX, [EBP - 4]        ;  faster above a certain threshold
MOV       [EAX + 8], EBX        ;  size of returned aggregate.

POP       ESI                   ; Begin Epilog by restoring
POP       EDI                   ;  preserved registers.
POP       EBX
MOV       ESP, EBP              ; Deallocate callee's local
POP       EBP                   ; Restore caller's EBP
RET                             ; Return to caller
```



Caller's code just after call:

```
ADD    ESP, 24       ; Remove parameters from stack
 ...                 ; Because address of y was given as the
                     ;  hidden parameter, the assignment of the
                     ;  return value has already been performed.
```

Examples Using _Optlink



If a `y.a = bar(x).b` construct is used instead of the more common `y = bar(x)` construct, the address of the return value is available in EAX. In this case, the address of the return value (hidden parameter) would point to a temporary variable allocated by the compiler in the automatic storage of the caller.

Passing and Returning Aggregates by Value to an Unprototyped Routine

This example differs from the previous one by the presence of an eyecatcher after the call to `bar` in the caller's code and the code necessary to perform the default widening rules required by ANSI.

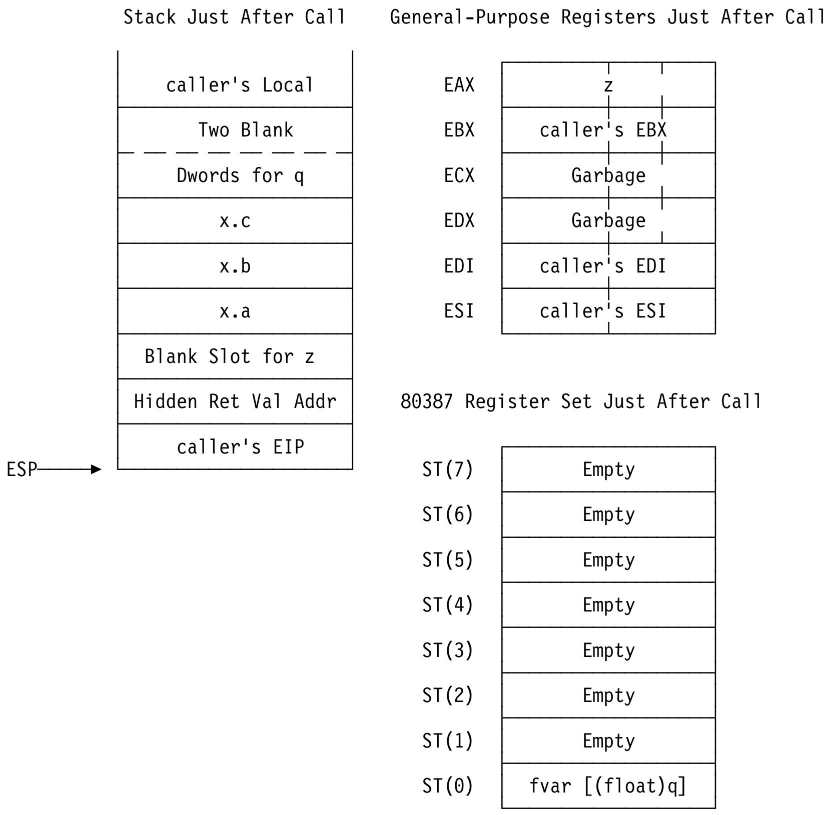
```
struct s_tag {
         long  a;
         float b;
         long  c;
         } x, y;
long  z;
double q;
```

```
/* Actual Call */
y = bar(z, x, q);
 ...

/* callee */
struct s_tag bar(long lvar, struct s_tag aggr, float fvar)
{
   struct s_tag temp;

   temp.a = lvar + aggr.a + 23;
   temp.b = fvar - aggr.b;
   temp.c = aggr.c

   return temp;
}
```

Caller's code up until call:

```
FLD     QWORD_PTR q         ; Load lexically first floating-point
                            ;  parameter to be converted
SUB     ESP, 8              ; Allocate space for the floating-point
                            ;  register parameter
PUSH    x.c                 ; Push nonconforming parameters on
PUSH    x.b                 ;  stack
PUSH    x.a                 ;
MOV     EAX, z              ; Load lexically first
                            ;  conforming parameter
                            ;  into EAX
SUB     ESP, 4              ; Allocate stack space for the first
                            ;  general-purpose register parameter.
PUSH    addr y              ; Push hidden first parameter (address of
                            ;  return space)
CALL    BAR
TEST    EAX, 00408000h      ; Eyecatcher
ADD     ESP, 28             ; Clean up parameters
```

_System Calling Convention



_System Calling Convention

To use this linkage convention, you must use the **_System** keyword in the declaration of the function or specify the `/Ms` option when you invoke the compiler.

Note: Because the VisualAge C++ library functions use the **_Optlink** convention, if you use the `/Ms` option, you must include all appropriate library header files to ensure the functions are called with the correct convention.

The following rules apply to the **_System** calling convention:

- All parameters are passed on the 80386 stack.
- The C parameter-passing convention is followed, where parameters are pushed onto the stack in right-to-left order.
- The calling function is responsible for removing parameters from the stack.
- All parameters are doubleword (4-byte) aligned.

- The size of the parameter list is passed in AL. If the parameter list is greater than 255 doublewords, the value contained in AL is the 8 least significant bits of the size. You can use the `__parmdwords` function (described in the *C Library Reference*) to access the value of AL that was passed to the function.

  Note: The `__parmdwords` function may not yield the correct value in the case of compilers that do not follow the rule of passing the size of the parameter list in AL. You can use the `__parmdwords` function with code compiled under IBM VisualAge C++ Version 3.0 for OS/2 provided the code was compiled with the `/Gp+` option specified.
- All functions returning non-floating-point values pass a return value back to the caller in EAX. Functions returning floating-point values use the floating-point stack ST(0). Aggregate return values, such as structures, are passed as a hidden parameter on the stack, and EAX points to them on return.
- All functions preserve the general purpose registers of the caller, except for ECX, EDX, and EAX.
- Structures passed by value are actually copied onto the stack, not passed by reference.
- The floating-point stack is defined to be empty upon entry to a called function, and has either a single item in ST(0) if there is a floating-point return, or is empty if there is not a floating-point return.
- The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.
- The compiler will not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

Examples Using the _System Convention

The following examples are included for purposes of illustration and clarity only. They have not been optimized. The examples assume that you are familiar with programming in assembler. Note that, in the examples, the stack grows toward the bottom of the page, and `ESP` always points to the top of the stack.

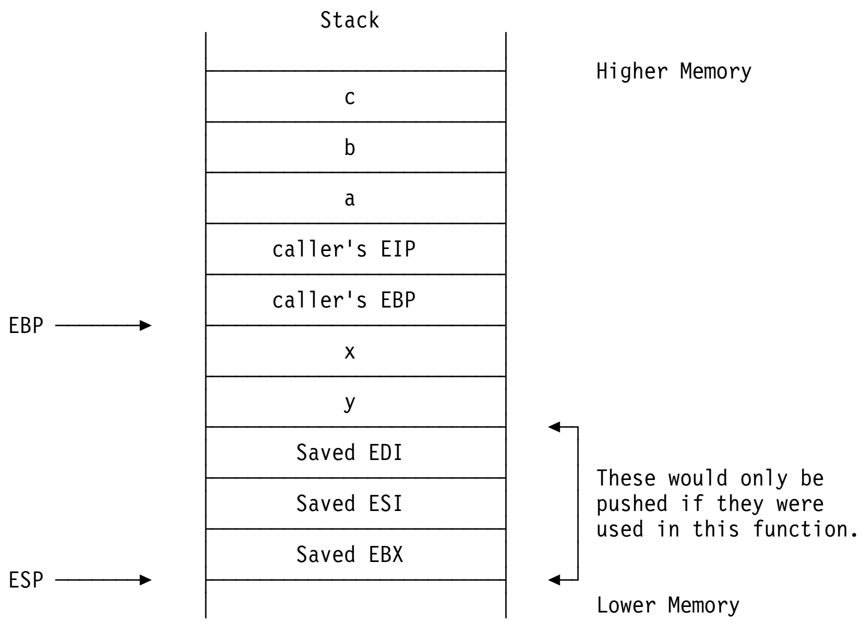
For the call

```
m = func(a,b,c);
```

`a`, `b`, and `c` are 32-bit integers and `func` has two local variables, `x` and `y` (both 32-bit integers).

Examples Using the _System Convention

The stack for the call to `func` would look like this:



The instructions used to create this activation record on the stack look like this on the calling side:

```
PUSH    c
PUSH    b
PUSH    c
MOV     AL, 3H
CALL    func
    .
    .
ADD     ESP, 12    ; Cleaning up the parameters
    .
    .
MOV     m, EAX
    .
    .
```

Note: the `MOV    AL,3H` instruction is not present unless the `/Gp` option is on.

For the callee, the code looks like this:

```
func PROC
   PUSH    EBP
   MOV     EBP, ESP       ;  Allocating 8 bytes of storage
   SUB     ESP, 8         ;   for two local variables.
   PUSH    EDI            ; These would only be
   PUSH    ESI            ;  pushed if they were used
   PUSH    EBX            ;  in this function.
   .
   .
   MOV     EAX, [EBP - 8]    ; Load y into EAX
   MOV     EBX, [EBP + 12]   ; Load b into EBX
   .
   .
   XOR     EAX, EAX           ; Zero the return value
   POP     EBX                ; Restore the saved registers
   POP     ESI
   POP     EDI
   LEAVE                      ; Equivalent to  MOV   ESP, EBP
                              ;                POP   EBP
   RET
func ENDP
```

The saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Floating-point results are returned in ST(0), which is the top of the floating-point register stack. If there is no numeric coprocessor installed in the system, the OS/2 operating system emulates the coprocessor.

Floating-point parameters are pushed on the 80386 stack.

Under some circumstances, the compiler will not use EBP to access automatic and parameter values, thus increasing the efficiency of the application. Whether it is used or not, EBP will not change across the call.

When passing structures as value parameters, the compiler generates code to copy the structure on to the 80386 stack. If the size of the structure is larger than an 80386 page size (4K), the compiler generates code to copy the structure backward. (That is, the last byte in the structure is the first to be copied.) This operation ensures that the OS/2 guard page method of stack growth will function properly in the presence of large structures being passed by value. Refer to the *User's Guide* for more information on stack growth.

Examples Using the _System Convention

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are 4 bytes farther away from EBP than they would be if no structure return were involved. The address of the returned structure is returned in EAX.

In the most common case, where the return from a function is simply assigned to a variable, the compiler merely pushes the address of the variable as the hidden parameter.[2] For example:

```
struct test_tag
       {
       int a;
       int some_array[100];
       } test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
   test_parm.a = 42;
   return test_parm;
}

int main(void)
{
   test_struct = test_function(test_struct);
   return test_struct.a;
}
```

2 Note that when the `/Gp` switch is on and this function calls the `__parmdwords` function, the value of AL is stored in a temporary variable in its prolog. This is done to ensure that the value cannot change before the call to `__parmdwords`.

The code generated for this program would be:

```
test_function  PROC
   PUSH    ESI
   PUSH    EDI
   MOV     DWORD PTR [ESP+0cH], 02aH     ; test_parm.a
   MOV     EAX, [ESP+08H]                ; Get the target of the return value
   MOV     EDI, EAX                      ; Value
   LEA     ESI, [ESP+0cH]                ; test_parm
   MOV     ECX, 065H
   REP MOVSD
   POP     EDI
   POP     ESI
   RET
test_function  ENDP

   PUBLIC  main
main  PROC
   PUSH    EBP
   MOV     EBP, ESP
   PUSH    ESI
   PUSH    EDI

   SUB     ESP, 0194H                  ; Adjust the stack pointer
   MOV     EDI, ESP
   MOV     ESI, OFFSET FLAT: test_struct
   MOV     ECX, 065H
   REP MOVSD                           ; Copy the parameter
   MOV     AL, 065H
   PUSH    OFFSET FLAT: test_struct    ; Push the address of the target
   CALL    test_function
   ADD     ESP, 0198H
```

Examples Using the _System Convention

```
   MOV     EAX, DWORD PTR test_struct  ; Take care of the return
   POP     EDI                         ;  from main
   POP     ESI
   LEAVE
   RET
main   ENDP
```

Note: The `MOV  AL, 065H` is not present unless the `/Gp` switch is on.

In a slightly different case, where only one field of the structure is used by the caller (as shown in the following example), the compiler allocates sufficient temporary storage in the caller's local storage area on the stack to contain a copy of the structure. The address of this temporary storage will be pushed as the target for the return value. Once the call is completed, the desired member of the structure can be accessed as an offset from EAX, as can be seen in the code generated for the example:

```
struct test_tag
       {
       int a;
       int some_array[100];
       } test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
   test_parm.a = 42;
   return test_parm;
}

int main(void)
{
   return test_function(test_struct).a;
}
```

The code generated for this example would be:

```
   PUBLIC  main
main   PROC
   PUSH    EBP
   MOV     EBP, ESP
   SUB     ESP, 0194H     ; Allocate space for compiler-generated
   PUSH    ESI            ;  temporary variable
   PUSH    EDI
   SUB     ESP, 0194H
   MOV     EDI, ESP
   MOV     ESI, OFFSET FLAT: test_struct
   MOV     ECX, 065H
   REP MOVSD
   LEA     EAX, [ESP+019cH]
   PUSH    EAX
   MOV     AL, 065H
   CALL    test_function
   ADD     ESP, 0198H
   MOV     EAX, [EAX]      ; Note the convenience of having the
   POP     EDI             ;  address of the returned structure
   POP     ESI             ;  in EAX
   LEAVE
   RET
main   ENDP
```

Again, the `MOV    AL,3H` instruction is not present unless the `/Gp` option is on.

_Pascal and _Far32_Pascal Calling Conventions

VisualAge C++ compiler provides both a **_Pascal** and a **_Far32 _Pascal** convention. The **_Far32 _Pascal** convention allows you to make calls between different code segments in code that runs at ring `0`, and is only valid when the `/Gr+` option is specified. The **_Pascal** conventions are most commonly used to create virtual device drivers, as described in Chapter 11, “Developing Virtual Device Drivers” on page 187.

Notes:

1. These **_Pascal** linkage conventions should not be confused with the 16-bit **_Far16 _Pascal** convention which is provided for 16-bit compatibility.
2. The **_Far32 _Pascal** convention is not available in C++ programs.

The **_Pascal** and **_Far32 _Pascal** conventions follow the same rules as the **_System** convention with these exceptions:

- Function names are converted to uppercase.
- Parameters are pushed in a left-to-right lexical order.
- The callee is responsible for cleaning up the parameters.
- Variable argument functions are not supported.
- The size of the parameter list is **not** passed in AL.

Important: The compiler does **not** convert 16-bit or 32-bit **_Pascal** function names to uppercase. The case of the function name in the call must match the case in the function prototype. Function names are, however, converted to uppercase in the object module to allow calls from assembler.

Examples Using the _Pascal Convention

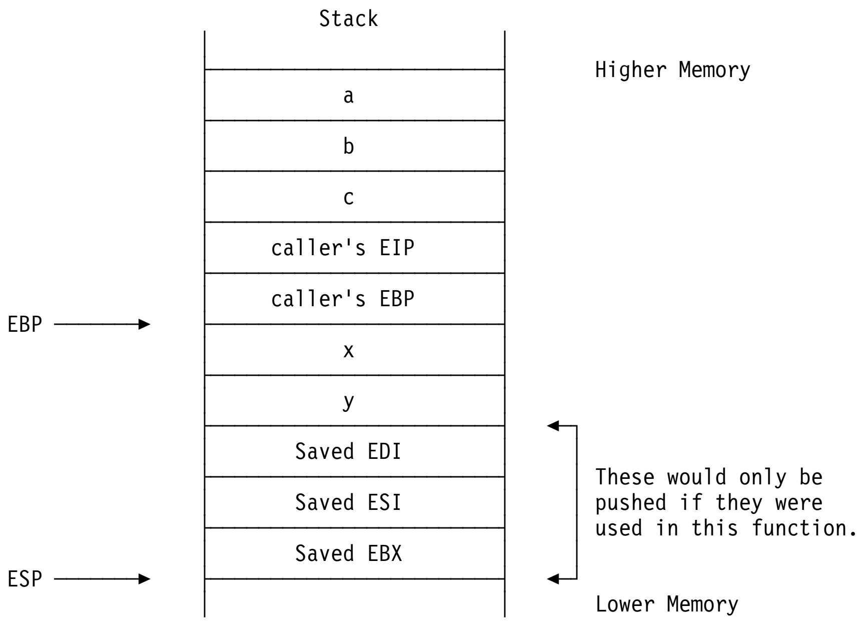
The following examples are included for purposes of illustration and clarity only and have not been optimized. The examples assume that you are familiar with programming in assembler. Note that, in the examples, the stack grows toward the bottom of the page, and `ESP` always points to the top of the stack.

For the call

```
m = func(a,b,c);
```

`a`, `b`, and `c` are 32-bit integers, and `func` has two local variables, `x` and `y` (both 32-bit integers).

The stack for the call to `func` would look like this:



The instructions used to build this activation record on the stack look like this on the calling side:

```
PUSH    a
PUSH    b
PUSH    c
CALL    FUNC
    .
    .
    .
MOV     m, EAX
    .
    .
```

Examples Using the _Pascal Convention

For the callee, the code looks like:

```
FUNC PROC
   PUSH    EBP
   MOV     EBP, ESP        ;  Allocating 8 bytes of storage
   SUB     ESP, 8          ;   for two local variables.
   PUSH    EDI             ; These would only be
   PUSH    ESI             ;  pushed if they were used
   PUSH    EBX             ;  in this function.
   .
   .
   MOV     EAX, [EBP - 8]    ; Load y into EAX
   MOV     EBX, [EBP + 12]   ; Load b into EBX
   .
   .
   XOR     EAX, EAX            ; Zero the return value
   POP     EBX                 ; Restore the saved registers
   POP     ESI
   POP     EDI
   LEAVE                       ; Equivalent to  MOV   ESP, EBP
                               ;                POP   EBP
   RET     0CH
FUNC ENDP
```

Like the **_System** calling convention, the saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Floating-point results are returned in ST(0). If there is no numeric coprocessor installed in the system, the OS/2 operating system emulates the coprocessor. Floating-point parameters are pushed on the 80386 stack.

_Far32 _Pascal function pointers are returned with the offset in EAX and the segment in DX.

In some circumstances, the compiler will not use EBP to access automatic and parameter values, thus increasing the efficiency of the application. Whether it is used or not, EBP will not change across the call.

Examples Using the _Pascal Convention

Structures are handled in the same way as they are under the **_System** calling convention. When passing structures as value parameters, the compiler generates code to copy the structure on to the 80386 stack. If the size of the structure is larger than an 80386 page size (4K), the compiler generates code to copy the structure backward. (That is, the last byte in the structure is the first to be copied.)

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are 4 bytes farther away from EBP than they would be if no structure return were involved. The address of the returned structure is returned in EAX.

In the most common case, where the return from a function is simply assigned to a variable, the compiler merely pushes the address of the variable as the hidden parameter. For example:

```
struct test_tag {
          int a;
          int some_array[100];
          } test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
   test_parm.a = 42;
   return test_parm;
}

int main(void)
{
   test_struct = test_function(test_struct);
   return test_struct.a;
}
```

The code generated for the above example would be:

Examples Using the _Pascal Convention

```
TEST_FUNCTION  PROC
   PUSH    EBP
   MOV     EBP, ESP
   PUSH    ESI
   PUSH    EDI
   MOV     DWORD PTR [ESP+0cH], 02aH   ; test_parm.a
   MOV     EAX, [EBP+08H]              ; Get the target of the return value
   MOV     EDI, EAX                    ; Value
   LEA     ESI, [EBP+0cH]              ; test_parm
   MOV     ECX, 065H
   REP MOVSD
   POP     EDI
   POP     ESI
   LEAVE
   RET     198H
TEST_FUNCTION  ENDP

   PUBLIC  main
main  PROC
   PUSH    EBP
   MOV     EBP, ESP
   PUSH    ESI
   PUSH    EDI

   SUB     ESP, 0194H                  ; Adjust the stack pointer
   MOV     EDI, ESP
   MOV     ESI, OFFSET FLAT: test_struct
   MOV     ECX, 065H
   REP MOVSD                           ; Copy the parameter
   PUSH    OFFSET FLAT: test_struct    ; Push the address of the target
   CALL    TEST_FUNCTION
   MOV     EAX, DWORD PTR test_struct  ; Take care of the return
   POP     EDI                         ;  from main
   POP     ESI
   LEAVE
   RET
main   ENDP
```

In a slightly different case, where only one field of the structure is used by the caller (as shown in the following example), the compiler allocates sufficient temporary storage in the caller's local storage area on the stack to contain a copy of the structure. The address of this temporary storage will be pushed as the target for the return value. Once the call is completed, the desired member of the structure can be accessed as an offset from EAX, as can be seen in the code generated for the example:

```
   struct test_tag {
             int a;
             int some_array[100];
             } test_struct;

   struct test_tag test_function(struct test_tag test_parm)
   {
      test_parm.a = 42;
      return test_parm;
   }

   int main(void)
   {
      return test_function(test_struct).a;
   }
```

The code generated for the example would be:

__stdcallCalling Convention

```
   PUBLIC  main
main   PROC
   PUSH    EBP
   MOV     EBP, ESP
   SUB     ESP, 0194H     ; Allocate space for compiler-generated
   PUSH    ESI            ;  temporary variable
   PUSH    EDI
   SUB     ESP, 0194H
   MOV     EDI, ESP
   MOV     ESI, OFFSET FLAT: test_struct
   MOV     ECX, 065H
   REP MOVSD
   LEA     EAX, [ESP+0194H]
   PUSH    EAX
   CALL    TEST_FUNCTION
   MOV     EAX, [EAX]      ; Note the convenience of having the
   POP     EDI             ;  address of the returned structure
   POP     ESI             ;  in EAX
   LEAVE
   RET
main   ENDP
```

__stdcall Calling Convention

The language details of these calling conventions are the same as for all other calling conventions. **__stdcall** has the additional restriction that an unprototyped **__stdcall** function with a variable number of arguments will not work.

To use this linkage convention, use the **`__stdcall`** keyword in the declaration of the function. You can make **__stdcall** the default linkage by specifying the `/Mt` option when you invoke the linker. There is no **`#pragma linkage`** for this convention.

The following rules apply to the **__stdcall** calling convention:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The *called* function removes the parameters from the stack.

- Floating point values are returned in `ST(0)` All functions returning non-floating point values return them in `EAX`, except for the special case of returning aggregates less than or equal to four bytes in size. For functions that return aggregates less than or equal to four bytes in size, the values are returned as follows:

Size of Aggregate	**Value Returned in**
4 bytes	EAX
3 bytes	EAX
2 bytes	AX
1 byte	AL

  For functions that return aggregates greater than four bytes in size, the address to place the return values is passed as a hidden parameter, and the addresss is passed back in `EAX`.

- Note that prototyped variable argument functions with **__stdcall** linkage are silently converted by the compiler to **__cdecl** linkage. Unprototyped functions may be given **__stdcall** linkage.

- Function names are decorated with an underscore prefix, and a suffix which consists of an at (@), followed by the number of bytes of parameters (in decimal). Parameters of less than four bytes are rounded up to four bytes. Structure sizes are also rounded up to a multiple of four bytes. For example, a function `fred` prototyped as follows:

  ```
  int fred(int, int, short);
  ```

  would appear as:

  ```
  _fred@12
  ```

  in the object module.

Note: When building export or import lists in `DEF` files, the decorated version of the name should be used. This is automatically handled when using **`#pragma export`** and **`#pragma import`**.

__cdecl Calling Convention

The **__cdecl** linkage is very similar to the OS/2 `_system` linkage convention. All general purpose registers are preserved except for EAX, ECX, and EDX. Note that, unlike the OS/2 `system` linkage, the number of dwords of parameters is **not** passed in AL. The `parmdwords` builtin function does not apply to **__cdecl**.

To use this linkage convention, use the **__cdecl** keyword in the declaration of the function. You can make **__cdecl** the default linkage by specifying the `/Mc` option when you invoke the linker. There is no **`#pragma linkage`** for this convention.

__cdeclCalling Convention

The following rules apply to the **__cdecl** calling convention:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The *calling* function removes the parameters from the stack.
- Floating point values are returned in `ST(0)`. All functions returning non-floating point values return them in `EAX`, except for the special case of returning aggregates less than or equal to four bytes in size. For functions that return aggregates less than or equal to four bytes in size, the values are returned as follows:

Size of Aggregate	**Value Returned in**
4 bytes	EAX
3 bytes	EAX
2 bytes	AX
1 byte	AL

  For functions that return aggregates greater than four bytes in size, the address to place the return values is passed as a hidden paramter, and the addresss is passed back in `EAX`.
- Function names are decorated with an underscore prefix when they appear in object modules. For example, a function named `fred` in the source program will appear as `_fred` in the object.

Note: When building export or import lists in `DEF` files, the decorated version of the name should be used. This is automatically handled when using **`#pragma export`** and **`#pragma import`**.

11

Developing Virtual Device Drivers

The VisualAge C++ compiler provides a number of features specifically for virtual device driver development. This chapter describes those features and discusses the issues you should be aware of when developing virtual device drivers. Note that support for developing virtual device drivers is available for C programs only.

Virtual device drivers (VDDs) provide virtual hardware support for DOS and DOS applications. They emulate input/output port and device memory operations. To achieve a certain level of hardware independence, a virtual device driver usually communicates with a physical device driver to interact with hardware. For example, the OS/2 operating system provides both virtual and physical device drivers for the mouse and keyboard.

User-supplied virtual device drivers simulate the hardware interfaces of an option adapter or device, and are usually used to migrate existing DOS applications into the OS/2 DOS environment.

A virtual device driver is essentially a DLL. It is responsible for presenting a virtual copy of the hardware resource to the DOS session and for coordinating physical access to that resource.

You may need to create a virtual device driver if multiple sessions must share access to a device where the input and output is not based on file handles, or if the particular device requires that interrupts be serviced within a short period of time.

Creating Code to Run at Ring Zero

Most object code runs at ring 3. However, some object code, such as that for virtual device drivers and operating systems, must run at ring `0`. To generate code to run at ring `0`, use the `/Gr+` option. Note that to use `/Gr+`, you must also specify the `/Rn` option and use the subsystem libraries.

When you use the `/Gr+` option, the compiler keeps track of which storage references are to the stack segment and which references are to the data segment, and ensures that the generated code is correct for these operations. This tracking is necessary because at ring `0`, the stack segment and data segment may not be the same. (At ring 3, they are the same.)

In some cases, the compiler cannot tell whether the reference is to the stack or data segment. Usually the reason is that the control flow of the program allows for either possibility, depending on which path through the program is taken at run time. For this reason, when you take the address of a stack-based variable (such as a local variable or parameter), you cannot safely pass the address to another function. In addition, you cannot safely store a stack address and a static or external address in the same variable, and subsequently de-reference the pointer created by the operation.

Whenever you take the address of a stack-based variable, the compiler generates a warning message that the address might be used in an unsafe way. This message is not generated if you specify `/Gr-`.

If your VDD contains any functions that are called from 16-bit physical device drivers, you must compile them with the `/Gv+` option to ensure the DS and ES registers are handled correctly. These two registers contain the selector for a 16-bit data segment. Using `/Gv+` ensures that DS and ES are saved on entry to an external function, set to the selector for DGROUP, and then restored on exit from the function.

Note: When you use `/Gv+`, if you also use the intermediate code linker (with the `/Fw+` or `/Ol+` option), only use the `/Gu+` option if the functions affected by `/Gv+` are explicitly exported. If they are not exported, do not use the `/Gu+` option. Because of this restriction, using the intermediate code linker for this type of program may not greatly improve the optimization of your code.

Using Virtual Device Driver Calling Conventions

If you are building a VDD in C, you must use 32-bit **_Pascal** or **_Far32 _Pascal** calling conventions to call the Virtual Driver Help interfaces or communicate with physical device drivers. These calling conventions are not supported for C++ programs. Within a VDD, you can use the **_Optlink** convention in most cases. Private interfaces between physical and virtual device drivers can use any calling convention provided both device drivers support it.

The **_Far32 _Pascal** calling convention is only available for code running at ring `0`. It allows you to make calls between code segments with different selectors. It also allows your VDDs to communicate with physical device drivers.

You can specify the calling convention using either the **_Pascal** and **_Far32 _Pascal** keywords. The description of the implementation of the **_Pascal** calling conventions is in "_Pascal and _Far32_Pascal Calling Conventions" on page 178.

Using _Far32 _Pascal Function Pointers

VisualAge C++ compiler provides special 48-bit function pointers so you can make indirect calls to 32-bit functions that use the **_Far32 _Pascal** convention. The **_Far32 _Pascal** pointers are required to build VDDs and similar applications that run at ring 0. For example, you would use 48-bit pointers to allow your VDD to communicate with physical device drivers.

The **_Far32 _Pascal** pointers, like the **_Far32 _Pascal** calling convention, are only supported when the /Gr+ option is specified.

The 48-bit pointer consists of 2 fields:

1. A 16-bit selector value which identifies the code segment.
2. A 32-bit offset value which identifies the function's location in the segment.

To declare a 48-bit pointer, use the **_Far32** and **_Pascal** keywords in the pointer declaration. For example:

```
void (* _Far32 _Pascal foo)(int);
```

declares foo to be a 48-bit pointer to a function with the **_Far32 _Pascal** convention that takes an integer argument and does not return a value.

The only operations that can be performed on or with a **_Far32 _Pascal** pointer are:

- Calling the function.
- Assigning the pointer, which includes casting it to a 32-bit function pointer or to an integer or unsigned type.
- Initializing the pointer, either statically or at runtime, with the address of a **_Far32 _Pascal** or 32-bit function, or with an integer or unsigned value.
- Comparing two pointers for equality or inequality. Like all function pointers, 48-bit pointers cannot be compared using relational operators.
- Passing the pointer as a parameter or returning it from a function. 48-bit pointers are passed in the same way as aggregates. The offset portion is returned in EAX and the selector portion in DX.

If you assign an integer or unsigned value to a 48-bit pointer, the selector field of the pointer is set to the default CODE32 segment, and the offset field is initialized to the integer value being assigned. This type of assignment is not generally useful, because you cannot know where a function will reside in a code segment, and because if your code segment is CODE32, a 32-bit function pointer is sufficient.

Note: **_Far32 _Pascal** pointers cannot be directly converted to **_Far16** pointers.

Creating a Module Definition File

When you link your VDD, you must use a module definition (`.DEF`) file. The first statement in the file must be

```
VIRTUAL DEVICE device_name
```

where *`device_name`* specifies the name of the VDD. The file cannot contain a `NAME` statement.

Once you have created your device driver, you must place a `DEVICE` statement in your CONFIG.SYS file to ensure it is treated as a device by the operating system.

For more details on `.DEF` file statements, see the *User's Guide*. For additional information on writing and building device drivers, see the online *Control Program Reference*.

12 Calling between 32-Bit and 16-Bit Code

If you have applications that depend on APIs that are only available as 16-bit code, or if you have developed or purchased libraries of routines that are currently 16-bit code, VisualAge C++ helps you protect that investment of time and money. Programmers can continue using their existing 16-bit source code and any 16-bit libraries on which their applications depend.

This chapter discusses how to:

- Call 16-bit code from your 32-bit VisualAge C++ programs
- Call 32-bit VisualAge C++ code from 16-bit code
- Pass data between 32-bit and 16-bit code.
- Share data between 32-bit and 16-bit code.

The conventions and methods described apply for both C and C++ programs.

Note: The VisualAge C++ compiler produces 32-bit code only. It does not produce 16-bit code.

Linking 32-bit and 16-bit Code

You can statically link between 32-bit and 16-bit code with the following restrictions:

- The `main` function must be 32-bit code.
- You cannot make any calls to 16-bit library functions in the 16-bit code.
- You must compile the 16-bit code with the /ND option (with a 16-bit compiler).

These restrictions do not apply when you dynamically link 32-bit code to 16-bit DLLs.

Calling 16-bit Code

There are three 16-bit calling conventions supported by VisualAge C++ compiler: **_Far16_Cdecl**, **_Far16 _Fastcall**, and **_Far16 _Pascal**.

The **_Far16_Cdecl** and **_Far16 _Pascal** conventions are equivalent to the `cdecl` and `pascal` conventions used in other compilers. The **_Far16 _Fastcall** convention is equivalent to the Microsoft C Version 6.0 `fastcall` convention.

Calling 16-bit Code

You can specify the calling convention for a function using keywords. For example, the following fragment uses keywords to declare the function dave as a 16-bit function using the **_Far16 _Pascal** calling convention:

```
void _Far16 _Pascal dave(short, long);
```

Similarities between the 16-Bit Conventions

The general rules for all three 16-bit calling conventions are:

- Types char, unsigned char, short, and unsigned short occupy a word on the stack.
- Types long and unsigned long occupy a doubleword with the value's high-order word pushed first.
- Types float, double, and long double are passed directly on the 80386 stack as 32-, 64-, and 80-bit values respectively.
- char types are sign-extended when expanded to word or doubleword size; unsigned char types are zero-extended on the stack.
- Far pointers are 32 bits and are pushed such that the segment value is pushed first and the offset second.
- If the argument is a structure, the last word is pushed first and each successive word is pushed until the first word.
- All arrays are passed by reference.
- BP, SI, and DI registers must be preserved across the call.
- Segment registers must be preserved across the call.
- Structures passed on the stack are rounded up in size to the next word boundary.
- The direction flag must be clear on entry and exit.
- Return values are passed back to the caller as follows:
  - Types char, unsigned char, short, and unsigned short are returned in AX.
  - Types long and unsigned long are returned such that the high word is in DX and the low word is in AX.
  - Far pointers are returned such that the offset is in AX and the selector is in DX.

Differences between the 16-Bit Conventions

- The order in which parameters are pushed on stack and their cleanup.

  When you use the **_Far16_Cdecl** calling convention, the parameters are pushed on the stack in a right-to-left order. The caller cleans up the parameters on the stack. This is the opposite of the **_Far16 _Pascal** and **_Far16 _Fastcall** conventions. When you use the **_Far16 _Pascal** convention, the parameters are pushed on the stack from left to right, and the callee (the function being called) cleans up the stack (usually by using a `RET nn` where `nn` is the number of bytes in the parameter list).

- The number of registers which can take parameters.

  The **_Far16 _Fastcall** convention differs from **_Far16_Cdecl** and **_Far16 _Pascal** in that it uses three registers that can take parameters, similar to **_Optlink**. When you use **_Far16 _Fastcall**, registers are assigned to variable types as follows:

  – Types `char` and `unsigned char` are stored in AL, DL, and BL.
  – Types `short` and `unsigned short` are stored in AX, DX, and BX.
  – Types `long` and `unsigned long` are stored such that the high word is in DX and the low word is in AX.
  – All other types are passed on the stack.

  Arguments are stored in the first available register allocated for their type. If all registers for that type are filled, the argument is pushed on the 80386 stack from left to right.

- The method of returning structures, unions, and floating point types.

  For **_Far16_Cdecl** and **_Far16 _Pascal**, all three types are returned with the address returned like a far pointer; that is, the value is in storage. The **_Far16 _Pascal** convention passes a hidden parameter, while **_Far16_Cdecl** has a static area. This means that the **_Far16_Cdecl** convention is nonreentrant, and should not be used in multithread programs. See “Return Values from 16-Bit Calls” on page 196 for more details on how values are returned from 16-bit calls.

  When you use the **_Far16 _Fastcall** convention, structures and unions are returned with the address returned like a near pointer. Like **_Far16 _Pascal**, **_Far16 _Fastcall** passes the address as a hidden parameter. Floating-point types are returned in ST(0).

Specifying Stack Size

You can specify the stack size for 16-bit code using the **`#pragma stack16`** directive. For example, the following directive sets the stack size to 8192 bytes (8K):

```
#pragma stack16(8192)
```

The default stack size is 4096 bytes (4K). This size is used for all 16-bit functions called after the **pragma** directive until the end of the compilation unit, or until another **#pragma stack16** is encountered. The 16-bit stack is allocated from the 32-bit stack, so you must ensure that the 32-bit stack is large enough for both your 32-bit and 16-bit code.

For more information on **#pragma stack16** and the linkage keywords, see the online *Language Reference*.

Compiler Option for 16-Bit Declarations

The VisualAge C++ compiler also provides the /Gt compiler option to enable data to be shared between 32-bit and 16-bit code. When you compile a program with /Gt+, an implicit **#pragma seg16** directive is performed for all variable declarations. Pointers are **not** implicitly qualified with **_Seg16**; you must qualify them if desired.

The /Gt+ option also defines special versions of the malloc family of functions that return memory that can be safely used by 16-bit code. When /Gt+ is specified, all calls to calloc, malloc, realloc, and free are mapped to _tcalloc, _tmalloc, _trealloc, and _tfree respectively.

These functions work exactly like the original functions, but the memory allocated or freed will not cross 64K boundaries. The objects declared can be used in 16-bit programs. This memory is also called *tiled* memory and is limited to 512M per process.

Note: When you use the /Gt+ option, data items larger than 64K in size will be aligned on 64K boundaries, but will also cross 64K boundaries.

Restrictions on 16-Bit Calls and Callbacks

- The compiler ensures that no parameters or automatic variables of a function calling 16-bit code cross a 64K boundary. Any parameters or automatic variables of functions that do not call 16-bit code may cross 64K boundaries. Passing the address of the parameters or automatic variables to functions that pass them on to 16-bit code will result in an unreliable program.

  To solve this problem, copy the value passed into an automatic variable in the function that calls the 16-bit code. This automatic variable will not cross a 64K boundary.

- Memory returned by _alloca will not be tiled. If a function contains a call to _alloca, it should not also call 16-bit code, because parameters and automatic variables may then cross 64K boundaries.

- A 16-bit program cannot pass structures by value to a 32-bit callback function. The callback function cannot return structures by value to the 16-bit program that called it.

- The parameter area of the callback function cannot be larger than 120 bytes.
- Callback functions that take a variable number of arguments are not supported.
- Calling 16-bit code that takes a variable number of arguments is not supported.

Example of Calling a 16-Bit Program

The sample program SAMPLE04 shows how to call 16-bit code from a 32-bit program, and also how to call back to a 32-bit function from a 16-bit routine. The 16-bit code is placed in two DLLs, one of which is bound to the 32-bit program at load time by using IMPLIB to build an import library. The other is bound at run time using OS/2 APIs. When the program is run, it prints a stanza from a poem.

Although the source for the 16-bit routines is included in SAMPLE04 for demonstration purposes, the mechanisms used to call the routines can also be applied when the 16-bit source is not available.

Important: To compile, link, and run this example, you must have the IBM C/2 or Microsoft C Version 6.0 16-bit compiler installed, and its main directory must be included in the PATH statement of your CONFIG.SYS file.

The files for the sample program are:

`SAMPLE04.C`	The source file for the 32-bit program
`SAMPLE04.H`	The user include file
`SAMPLE04.DEF`	The module definition file for the 32-bit program
`SAMP04A.C`	The source file for the first 16-bit DLL (bound at load time)
`SAMP04A.DEF`	The module definition file for the first 16-bit DLL.
`SAMP04B.C`	The source file for the second 16-bit DLL (bound at run time)
`SAMP04B.DEF`	The module definition file for the second 16-bit DLL.

The 32-bit main program (`SAMPLE04.C`):

- Makes direct calls to the 16-bit functions `plugh1` and `plugh2`, which are both defined in the 16-bit DLL bound at load time (the source for which is `SAMP04A.C`).
- Demonstrates a callback function. The 32-bit user function `xyzzy` is passed to the 16-bit `plugh3` routine (defined in `SAMP04A.C`) with the intent that the 16-bit routine will then call the user function. The `xyzzy` function is declared using a 16-bit calling convention and is called from the 16-bit DLL, but it is run as a 32-bit function.
- Uses OS/2 APIs to load the runtime DLL (the source for which is `SAMP04B.C`) and query the address of the function `plugh4`. The program then calls `plugh4` using the function pointer returned by the API.

If you installed the Sample programs, you will find the `SAMPLE04` project in the VisualAge C++ `Samples` folder. For information on how to build and debug a project, see the *User's Guide*.

Alternatively, if you wish to compile, link, and run the sample from the command line, you will find a `readme` file with instructions. in the `\IBMCPP\SAMPLES\COMPILER\SAMPLE04` directory along with the files needed which include two makefiles that build the sample. One makefile for static linking and one for dynamic linking.

Return Values from 16-Bit Calls

The following examples demonstrate how the VisualAge C++ compiler expects values to be returned from calls to 16-bit programs.

Note: This is the same way that the IBM C/2 and Microsoft C Version 6.0 compilers return values.

-
  ```
  char cdecl myfunc(double,float,struct x);
  char pascal myfunc(double,float,struct x);
  char fastcall myfunc(double,float,struct x);

  unsigned char cdecl myfunc(double,float,struct x);
  unsigned char pascal myfunc(double,float,struct x);
  unsigned char fastcall myfunc(double,float,struct x);
  ```

  The returned value is placed in AL.

-
  ```
  short cdecl myfunc(double,float,struct x);
  short pascal myfunc(double,float,struct x);
  short fastcall myfunc(double,float,struct x);
  ```

  The returned value is placed in AX.

-
  ```
  long cdecl myfunc(double,float,struct x);
  long pascal myfunc(double,float,struct x);
  long fastcall myfunc(double,float,struct x);
  ```

  The high word is in DX, and the low word is in AX.

-
  ```
  float cdecl myfunc(double, float, struct x);
  double cdecl myfunc(double, float, struct x);
  long double cdecl myfunc(double, float, struct x);
  ```

  The compiler does not provide a hidden parameter, but rather places the return value in an external static variable `__fac`, which is defined as a `QWORD`. On return, DX contains the selector and AX contains the offset of `__fac`.

  For functions with type `long double cdecl`, the returned value is placed in ST(0).

-
  ```
  float pascal myfunc(double,float,struct x);
  double pascal myfunc(double,float,struct x);
  long double pascal myfunc(double,float,struct x);
  ```

  The compiler reserves space in automatic storage for the return value and pushes (last) a pointer to this area (offset only, SS is always assumed). The callee stores the return value in this area and returns the offset of this area in AX and returns SS in DX.

-
  ```
  float fastcall myfunc(double,float,struct x);
  double fastcall myfunc(double,float,struct x);
  long double fastcall myfunc(double,float,struct x);
  ```

  The returned value is placed in ST(0).

-
  ```
  char far * cdecl myfunc(double,float,struct x);
  char far * pascal myfunc(double,float,struct x)
  char far * fastcall myfunc(double,float,struct x)
  ```

  Far pointers are returned such that the offset is in AX and the selector is in DX.

-
  ```
  struct_20_bytes cdecl myfunc(double,float,struct x)
  ```

  The compiler reserves `sizeof(struct_20_bytes)` in uninitialized static (BSS) for the callee. No hidden parameter is passed; the callee moves the return structure into this static reserved area and returns the offset of the structure in AX and the selector in DX.

- `struct_20_bytes pascal myfunc(double,float,struct x)`
  `struct_20_bytes fastcall myfunc(double,float,struct x)`

  The compiler reserves space for the return value in the caller's automatic storage and pushes the address of this area as a near pointer. SS will be assumed as the selector. This parameter is pushed last as a hidden parameter. The offset of the reserved space is returned in AX, and the selector (SS) is returned in DX.

- `struct_4_bytes cdecl myfunc(double,float,struct x)`
  `struct_4_bytes fastcall myfunc(double,float,struct x)`

  The compiler returns the contents of the structure in AX and DX. AX contains the lower 2 bytes, and DX the higher 2 bytes.

  – If the structure is packed and its size is 1 byte, AL is used.
  – If the structure's size is 2 bytes, AX is used.
  – If the structure is packed and its size is 3 bytes, space is reserved in the data segment, the offset is returned in AX, and the selector is returned in DX.

- `struct_4_bytes pascal myfunc(double,float,struct x)`

  The compiler reserves space for the return value in the caller's automatic storage and pushes the address of this area as a near pointer. SS will be assumed as the selector. This parameter is pushed last as a hidden parameter. The offset of the reserved space is returned in AX, and the selector (SS) is returned in DX.

- `char cdecl myfunc(double,float,struct x)`
  `char pascal myfunc(double,float,struct x)`
  `char fastcall myfunc(double,float,struct x)`

  `unsigned char cdecl myfunc(double,float,struct x)`
  `unsigned char pascal myfunc(double,float,struct x)`
  `unsigned char fastcall myfunc(double,float,struct x)`

  The returned value is placed in AL.

Calling Back to 32-bit Code from 16-bit Code

Some 16-bit applications require that calling applications register callback functions. For example, IBM Communications Manager requires callback functions to handle some events. When you call these 16-bit applications from 32-bit code, you can pass a pointer to a 32-bit function that will act as the callback function.

The 32-bit callback function must use the **_Far16_Cdecl** or **_Far16 _Pascal** calling convention. The **_Far16 _Fastcall** convention is not supported for callback functions. All pointer parameters must be qualified with the **_Seg16** type qualifier.

The VisualAge C++ compiler performs all necessary changes from the 16-bit to the 32-bit environment on entry to the callback function, and from 32-bit to 16-bit on exit.

Passing Data between 16-bit and 32-bit Code

If a structure will be referenced in both 32-bit and 16-bit code and contains bit-fields or members of type `int` or `enum`, you may have to rewrite the structure to ensure that all members align properly.

int
To ensure all integers map the same way, change your integer declarations to use `short` for 2-byte integers and `long` for 4-byte integers.

long double
To use your IBM C/2 `long double` under VisualAge C++ you must declare it as `double`.

If you wish to use Microsoft C Version 6.0 `long double` data under VisualAge C++, you must recompile it on the Microsoft C Version 6.0 compiler as `double` first, and place into into a struct to pad out to 128 bits. Although, VisualAge C++ and Microsoft C Version 6.0 both store `long double` as 80 bit real, VisualAge C++ stores it in a 16 byte (128 bit) field.

enum
Use the `/Su2` option to force `enum` variables to be 2 bytes in size. This will make them compatible with 16-bit code.

The size of type `enum` differs between compilers. For example, the IBM C/2 makes all `enum` types 2 bytes, while the VisualAge C++ compiler defines the size as 1, 2, or 4 bytes, depending on the range of values the enumeration contains.

You can use the `/Su` option to force the VisualAge C++ compiler to make the `enum` type 1,2, or 4 bytes, or to use the SAA rules that make all `enum` types the size of the smallest integral type that can contain all variables.

bit fields
VisualAge C++ and 16-bit compilers use entirely different algorithms for packing bit fields. In general, if the bit fields are packed "tightly", they will be compatible. "Tightly" means that there is no padding introduced by the compiler. IBM C/2 and Microsoft C Version 6.0 use the type specifier for the bit field member to determine the size of the bit field area. VisualAge C++ allocates the minimum number of bytes possible.

Passing Data between 16-bit and 32-bit Code

```
struct s1 {          /* Compatible */     1
  int a : 4;
  int b : 7;
  int c : 5;
};

struct s2 {          /* Not compatible */ 2
  int z : 9;
  int y : 12;
};
```

1 Because the type of the bit field members is `int`, Microsoft C Version 6.0 will allocate 16 bits' (`sizeof(int)`) of space in the `struct` to store the bit field.

There are 16 bits of bit fields declared in the declaration of `s1`, so the bit fields are packed tightly.

VisualAge C++ allocates precisely 2 bytes for the bit field., Microsoft C Version 6.0 and VisualAge C++ lay out the bit field in precisely the same way. Thus, they are compatible.

2 There is not enough room in the 16-bit `int` to store the first two members of `s2`, so Microsoft C Version 6.0 introduces 7 bits of padding to made the member `z` fill out an entire `int`. It introduces an additional 4 bits of padding after `y` to fill up the last byte. This bit field requires 4 bytes of storage using Microsoft C Version 6.0, but only 3 bytes using VisualAge C++. Therefore, the padding introduced by the compiler makes the bit fields incompatible.

Compilers utilizing 16-bit format align these types on 1 byte boundaries.

VisualAge C++ aligns members of type `int`, `long`, `float`, `double`, `long double`, and `pointer` on 4-byte boundaries.

structure layout
Use `/Sp1` or **`#pragma pack(1)`** on structures declarations that will be passed to or from 16-bit code.

segmented pointers
If the pointer is passed to the function as a member of an aggregate or an array, you must qualify it with **_Seg16**. The **_Seg16** keyword tells the compiler to store the pointer as a segmented pointer and not as a flat pointers used in 32-bit code. By segmented, we mean a 16-bit segment selector and a 16-bit offset. The **_Seg16** keyword is also required if you are using two or more levels of indirection (for example, a pointer to a pointer).

If the pointer is passed directly as a parameter, the compiler automatically converts it to a 16-bit pointer and the **_Seg16** keyword is not required. For example, the declaration

```
void _Far16 _cdecl foo(char *);
```

is equivalent to

```
void _Far16 _cdecl foo(char * _Seg16);
```

It is good programming style to explicitly declare pointer parameters in 16-bit function prototypes as being **_Seg16**.

If your pointers are used primarily as parameters to 16-bit functions and are not used extensively in your 32-bit code, it may be advantageous to declare them with **_Seg16**.

Use the **_Seg16** qualifier only when necessary. Because of the conversions that are performed whenever a **_Seg16** pointer is used in 32-bit code, unnecessary use of segmented pointers can cause a noticeable degradation in the performance of your application.

Sharing Data between 32-bit and 16-bit Code

Declaring Segmented Pointers

Use the **_Seg16** type qualifier to declare external pointers that will be shared between 32-bit and 16-bit code, that is, that are declared in both.

For example:

```
char * _Seg16 p16;
```

directs the compiler to store the pointer as a segmented pointer (with a 16-bit selector and 16-bit offset) that can be used directly by a 16-bit application. The **_Seg16** keyword comes **after** the asterisk in the declaration, as required by ANSI syntax rules.

When a **_Seg16** pointer is used in 32-bit code, the VisualAge C++ compiler automatically converts it to a flat 32-bit pointer when necessary.

```
char * _Seg16 p16;
char * _Seg16 * _Seg16 pp16;
char * p32;

p32=p16;        1          /* Automatic conversion */
*pp16=p32;      2          /*           Here, too  */
p16++;          3     /* Two conversions happen here */
```

1 p16 is converted from seg to flat before being stored in p32.

2 pp16 is converted to flat before being dereferenced, p32 is converted to seg before being stored in *pp16.

3 p16 is converted to flat, is incremented, and then converted back to seg before being stored back in p16.

Note: The **_Seg16** keyword comes **after** the asterisk in the declaration, as required by ANSI syntax rules. Programmers familiar with other compilers may be accustomed to placing the far keyword in their declarations, but to the left of the asterisk:

```
char far * x;
```

Because this syntax is contrary to ANSI binding rules, VisualAge C++ product does not support it.

Declaring Shared Objects

Because a 16-bit program cannot access a data item that is larger than 64K in size or that spans a 64K boundary in memory, any data items that are to be shared between 16-bit and 32-bit programs must conform to these limits. Use the **#pragma seg16** directive to ensure that shared data items do not cross 64K boundaries. In most cases, you need only use this **#pragma** directive with items that are likely to cross 64K boundaries, such as aggregates, doubles, and long doubles.

You can use **#pragma seg16** either with the data item directly or through a typedef. The following code fragment shows both ways of using **#pragma seg16**:

```
struct family {
  long          john;
  double        carolynn;
  char * _Seg16 geoff;
  long          colleen;
};

#pragma seg16( cat )
struct family cat;          /* cat is qualified directly */

typedef struct family tom;        1
#pragma seg16( tom )              2

tom  edna;        /* edna is qualified using a typedef */  3
```

Note: Using **#pragma seg16** on variables of type struct family does not mean that pointers inside the structure will automatically be qualified with **_Seg16**. If you want the pointers to be qualified as such, you must declare them yourself.

The **`#pragma seg16`** directive can be used either before or after the variable or `typedef` name is declared. In the case of the `typedef`, however, the **#pragma** must be attached to the `typedef` name before that name is used in another declaration. For example, in the preceding example, the lines marked **1** and **2** can appear in any order, but both must appear before the line marked **3**.

Because data objects used in 16-bit programs must be smaller than 64K, the **`#pragma seg16`** directive cannot be used on objects greater than 64K.

Declaring Objects Shared Objects

13

Developing Subsystems

A subsystem is a collection of code and/or data that can be shared across processes and that does not use the VisualAge C++ runtime environment. This chapter describes how to create a subsystem.

A subsystem may have code and data segments that are shared by all processes, or it may have separate segments for each process. If the subsystem is a DLL, there is also an initialization routine associated with it.

By default, VisualAge C++ compiler creates a runtime environment for you using C or C++ initializations, exception management, and termination. This environment allows runtime functions to perform input/output and other services. However, many applications require no runtime environment and must be written as subsystems. For example, you will want to turn off the runtime environment support to:

- Develop Presentation Manager display or printer drivers
- Develop virtual device drivers
- Develop installable file system drivers
- Create DLLs with global initialization/termination and a single automatic data segment that is shared by all processes. The initialization/termination function is called once when the DLL is first loaded and once more when it is last freed.

Creating a Subsystem

To create a subsystem, you must first create one or more source files as you would for any other program. Subsystems can be written in C or C++. No special file extension is required.

When you do not use the runtime environment, you must provide your own initialization functions, multithread support, exception handling, and termination functions. You can use OS/2 APIs. For more information on the OS/2 APIs, see the *Control Program Guide and Reference* and the *PM Guide and Reference*.

If you need to pass parameters to a subsystem executable module, the `argv` and `argc` command-line parameters to `main` are supported. However, you cannot use the `envp` parameter to `main`.

Subsystem Library Functions

The libraries `CPPON30.LIB` and `CPPON30.DLL` are provided specifically for subsystem development. Use `CPPON30.LIB` for static linking, and `CPPON30.DLL` for dynamic linking. The import library `CPPON30I.LIB` is also provided for dynamic linking. You can also use the `CPPON30O.LIB` library to create your own subsystem runtime DLL. See "Creating Your Own Subsystem Runtime Library DLLs" on page 213 for more information on creating subsystem runtime DLLs.

Those VisualAge C++ library functions that require a runtime environment cannot be used in a subsystem. The subsystem libraries contain the library functions that do not require a runtime environment, including the extensions that allow low-level I/O. No other I/O functions are provided.

With the exception of the memory allocation functions (`calloc, malloc, realloc`, and `free`), all of the functions in the subsystem libraries are reentrant.

Note: Although the low-level I/O functions defined in **<io.h>** are reentrant, you should serialize access to these functions within each file. If you do not serialize the access, you may get unexpected input or output.

The C++ runtime functions (`new` and `delete`) and exception handling functions (`throw, try` and `catch`) are also available for subsystem development. None of the Open Classes are are available for subsystem development.

There are three groups of functions that you can use in a subsystem:

1. The subsystem library functions listed below. These functions are available whether or not you have optimization turned on (`/O+`).
2. Built-in instrinsic functions. These are listed in "Functions that Are Always Inlined" on page 358. These functions are also available whether or not you have optimization turned on.
3. Other intrinsic functions. These are listed in "Functions that Are Inlined when Optimization Is On" on page 357. These functions are **only** available for use in a subsystem if optimization is turned on.

Subsystem Library Functions

The functions available in the subsystem libraries are:

`abs`	`__eof`	`qsort`	`_tell`
`access`	`exit`[2]	`read`	`_uaddmem`
`atof`	`_filelength`	`realloc`	`_ucalloc`
`atoi`[1]	`_fpreset`	`realloc`	`_uclose`
`atol`[1]	`free`	`_set_crt_msg_handle`	`_ucreate`
`bsearch`	`_heap_check`	`setjmp`[3]	`_udefault`
`calloc`	`_heap_walk`	`_setmode`	`_udestroy`
`chmod`	`_heapchk`	`_sopen`	`_udump_allocated`
`_chsize`	`_heapmin`	`sprintf`[4]	`_udump_allocated_delta`
`_clear87`	`_heapset`	`sscanf`[4]	`_uheap_check`
`close`	`isatty`	`_status87`	`_uheap_walk`
`_control87`	`_itoa`	`strcat`	`_uheapchk`
`creat`	`labs`	`strchr`	`_uheapmin`
`_debug_calloc`	`ldiv`	`strcmp`	`_uheapset`
`_debug_free`	`longjmp`[3]	`strcpy`	`_ultoa`
`_debug_heapmin`	`lseek`	`strcspn`	`_umalloc`
`_debug_malloc`	`_ltoa`	`strdup`	`umask`
`_debug_realloc`	`malloc`	`strncat`	`_uopen`
`_debug_ucalloc`	`memchr`	`strncmp`	`_ustats`
`_debug_uheapmin`	`memcmp`	`strncpy`	`va_arg`[5]
`_debug_umalloc`	`memcpy`	`strpbrk`	`va_end`[5]
`div`	`memmove`	`strrchr`	`va_start`[5]
`_dump_allocated_delta`	`memset`	`strspn`	`vprintf`[4]
`_dump_allocated`	`_mheap`	`strstr`	`vsprintf`[4]
`dup`	`_msize`	`strtol`	`write`
`dup2`	`open`	`strtoul`	
	`printf`[4]		

Notes:

1. The subsystem library versions of these functions do not use the locale information that the standard library versions use.
2. Note that `atexit` and `_onexit` are not provided.
3. You must write your own exception handler when using these functions in a subsystem.
4. When you use these functions in a subsystem, `\n` will be translated to `\r\n` and `DosWrite` will be used to write the contents of the buffer to `stdout`. There is no serialization protection and no multibyte support. These functions use only the default "C" locale information.
5. These functions are implemented as macros.

Calling Conventions for Subsystem Functions

When creating a subsystem, you can use either the **_System** or **_Optlink** calling convention for your functions. Any external functions that will be called from programs not compiled by the VisualAge C++ compiler **must** use the **_System** convention.

You can use the `/Mp` or `/Ms` options to specify the default calling convention for all functions in a program, and you can use linkage keywords or the **`#pragma linkage`** directive to specify the convention for individual functions.

Note: The **`#pragma linkage`** directive is supported for C programs only.

Building a Subsystem DLL

To create a subsystem DLL, follow the steps described in Chapter 6, "Building Dynamic Link Libraries" on page 61. The steps are the same as for a DLL that uses the runtime environment.

The one difference between the two types of DLLs is the `_DLL_InitTerm` function. This function is the initialization and termination entry point for all DLLs. In the C runtime environment, `_DLL_InitTerm` initializes and terminates the necessary environment for the DLL, including storage, semaphores, and variables. The version provided in the subsystem libraries defines the entry point for the DLL, but provides no initialization or termination functions.

If your subsystem DLL requires any initialization or termination, you will need to create your own `_DLL_InitTerm` function. Otherwise, you can use the default version.

Writing Your Own Subsystem `_DLL_InitTerm` Function

The prototype for the `_DLL_InitTerm` function is:

```
unsigned long _System _DLL_InitTerm(unsigned long hModule,
                                    unsigned long ulFlag);
```

If the value of the *ulFlag* parameter is 0, the DLL environment is initialized. If the value of the *ulFlag* parameter is 1, the DLL environment is ended.

The *hModule* parameter is the module handle assigned by the operating system for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, `DosQueryModuleName` can be used to return the fully qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from `_DLL_InitTerm` tells the loader if the initialization or termination 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.

Because it is called by the operating system loader, the _DLL_InitTerm function must be declared as having the **_System** calling convention.

You do not need to call _CRT_init and _CRT_term in your _DLL_InitTerm function, because there is no runtime environment to initialize or terminate. However, if you are coding in C++, you do need to call __ctordtorInit at the beginning of _DLL_InitTerm to correctly initialize static constructors and destructors, and __ctordtorTerm at the end to correctly terminate them.

If you change your DLL at a later time to use the regular runtime libraries, you must add calls to _CRT_init and _CRT_term, as described in "Writing Your Own _DLL_InitTerm Function" on page 77, to ensure that the runtime environment is correctly initialized.

Example of a Subsystem _DLL_InitTerm Function

The following figure shows the _DLL_InitTerm function for the sample program SAMPLE05. If you installed the Sample programs, you will find the SAMPLE05 project in the VisualAge C++ Samples folder. For information on how to build and debug a project, see the *User's Guide*. Alternatively, if you wish to compile, link, and run the sample from the command line, you will find a readme file with instructions. in the \IBMCPP\SAMPLES\COMPILER\SAMPLE05 directory along with the files needed. This _DLL_InitTerm function is included in the SAMPLE05.C source file. You could also make your _DLL_InitTerm function a separate file. Note that this figure shows only a fragment of SAMPLE05.C and not the entire source file.

```
/* _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 )
   {
   APIRET rc;

   /* 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 ( ( rc = DosCreateMutexSem( SHARED_SEMAPHORE_NAME,
                                           &hmtxSharedSem,
                                           0,
                                           FALSE ) ) != NO_ERROR )
               {
               printf( "DosCreateMutexSem rc = %lu\n", rc );
               return 0;
               }
            }
```

Figure 16 (Part 1 of 2). _DLL_InitTerm *Function for* SAMPLE05.

```
         /* Register the current process.                                      */

         if ( DLLREGISTER( ) )
            return 0;

         break;

      case 1:
         /* De-register the current process.                                   */

         if ( DLLDEREGISTER( ) )
            return 0;

         _rmem_term();

         break;

      default:
         return 0;
      }

   /* Indicate success.  Non-zero means success!!!                             */

   return 1;
   }
```

Figure 16 (Part 2 of 2). `_DLL_InitTerm` *Function for* `SAMPLE05`.

Compiling Your Subsystem

To compile your source files into a subsystem, use the `/Rn` compiler option. When you use this option, the compiler does not generate the external references that would build an environment. The subsystem libraries are also specified in each object file to be linked in at link time. The default compiler option is `/Re`, which creates an object with a runtime environment.

If you are creating a subsystem DLL, you must use the `/Ge-` option in addition to `/Rn`. You can use either static linking (`/Gd-`), which is the default, or dynamic linking (`/Gd+`).

Restrictions When You Are Using Subsystems

If you are creating an executable module, the envp parameter to main is not supported. However, the argv and argc parameters are available. See "Passing Data to a Program" on page 13 for a description of envp under the runtime environment.

The low-level I/O functions allow you to perform some input and output operations. You are responsible for the buffering and formatting of I/O.

Example of a Subsystem DLL

The sample program SAMPLE05 shows how to create a simple subsystem DLL and a program to access it.

The DLL keeps a global count of the number of processes that access it, running totals for each process that accesses the subsystem, and a grand total for all processes. There are two external entry points for programs accessing the subsystem. The first is DLLINCREMENT, which increments both the grand total and the total for the calling process by the amount passed in. The second entry point is DLLSTATS, which prints out statistics kept by the subsystem, including the grand total and the total for the current process.

The grand total and the total for the process are stored in a single shared data segment of the subsystem. Each process total is stored in its own data segment.

The files for the sample program are:

SAMPLE05.C	The source file to create the DLL.
SAMPLE05.DEF	The module definition file for the DLL.
SAMPLE05.H	The user include file.
MAIN05.C	The main program that accesses the subsystem.
MAIN05.DEF	The module definition file for MAIN05.C.

If you installed the Sample programs, you will find the SAMPLE05 project in the VisualAge C++ Samples folder. For information on how to build and debug a project, see the *User's Guide*. Alternatively, if you wish to compile, link, and run the sample from the command line, you will find a readme file with instructions. in the \IBMCPP\SAMPLES\COMPILER\SAMPLE05 directory along with the files needed.

Creating Your Own Subsystem Runtime Library DLLs

If you are shipping your application to other users, you can use one of three methods to make the VisualAge C++ subsystem library functions available to the users of your application:

1. Statically bind every module to the library (`.LIB`) files.

   This method increases the size of your modules and also slows the performance because the DLL environment has to be initialized for each module.

2. Use the DLLRNAME utility to rename the VisualAge C++ subsystem library DLLs.

   You can then ship the renamed DLLs with your application. DLLRNAME is described in the *User's Guide*.

3. Create your own runtime DLLs.

   This method provides one common DLL environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the VisualAge C++ DLLs change, you need only rebuild your own DLL. In addition, you can tailor your runtime DLL to contain only those functions you use, including your own.

To create your own subsystem runtime library, follow these steps:

1. Copy and rename the VisualAge C++ `CPPON30.DEF` file, for example to `mysdll.def`. You must also change the DLL name on the `LIBRARY` line of the `.DEF` file. `CPPON30.DEF` is installed in the LIB subdirectory under the main VisualAge C++ installation directory.
2. Remove any functions that you do not use directly or indirectly (through other functions) from your `.DEF` file (`mysdll.def`), including the `STUB` line. Do not delete anything with the comment `****` next to it; variables and functions indicated by this comment are used by startup functions and are always required.

3. Create a source file for your DLL, for example, `mysdll.c`. If you are creating a runtime library that contains only VisualAge C++ functions, create an empty source file. If you are adding your own functions to the library, put the code for them in this file.

4. Compile and link your DLL files. Use the `/Ge-` option to create a DLL and the `/Rn` option to create a subsystem. For example:

   ```
   icc /Ge- /Rn mysdll.c mysdll.def
   ```

5. Use the IMPLIB utility to create an import library for your DLL, as described in "Using Your DLL" on page 72. For example:

   ```
   IMPLIB /NOI mysdlli.lib mysdll.def
   ```

6. Use the ILIB utility to add the object modules that contain the initialization and termination functions to your import library. These objects are needed by all executable modules and DLLs, and are contained in `CPPON30O.LIB` for subsystem programs. See the *User's Guide* for information on how to use ILIB.

   Note: If you do not use the ILIB utility, you must ensure that all objects that access your runtime DLL are statically linked to the appropriate object library. The compile and link commands are described in the next step.

7. Compile your executable modules and other DLLs with the `/Gn+` option to exclude the default library information. For example:

   ```
   icc /C /Gn+ /Ge+ /Rn myprog.c
   icc /C /Gn+ /Ge- /Rn mydll.c
   ```

   When you link your objects, specify your own import library. If you are using or plan to use OS/2 APIs, specify OS2386.LIB also. For example:

   ```
   ILINK myprog.obj mysdlli.lib OS2386.LIB
   ILINK mydll.obj mysdlli.lib OS2386.LIB
   ```

   To compile and link in one step, use the commands:

   ```
   icc /Gn+ /Ge+ /Rn myprog.c mysdlli.lib OS2386.LIB
   icc /Gn+ /Ge- /Rn mydll.c mysdlli.lib OS2386.LIB
   ```

Note: If you did not use the ILIB utility to add the initialization and termination objects to your import library, when you link your modules, specify:

a. `CPPON300.LIB`
b. Your import library
c. OS2386.LIB (to allow you to use OS/2 APIs)
d. The linker option `/NOD`.

For example:

```
ILINK /NOD myprog.obj CPPON300.LIB mysdlli.lib OS2386.LIB;
ILINK /NOD mydll.obj CPPON300.LIB mysdlli.lib OS2386.LIB;
```

The `/NOD` option tells the linker to disregard the default libraries specified in the object files and use only the libraries given on the command line. If you are using `icc` to invoke the linker for you, the commands would be:

```
icc /B"/NOD" /Rn myprog.c CPPON300.LIB mysdlli.lib OS2386.LIB
icc /Ge- /B"/NOD" /Rn mydll.c CPPON300.LIB mysdlli.lib OS2386.LIB
```

The linker then links the objects from the object library directly into your executable module or DLL.

Creating Subsystem Runtime Library DLLs

14 Signal and OS/2 Exception Handling

VisualAge C++ product and the OS/2 operating system both have the capability to detect and report runtime errors and abnormal conditions.

Abnormal conditions can be reported to you and handled in one of the following ways:

1. Using VisualAge C++ signal handlers. Error handling by signals is defined by the SAA and ANSI C standards and can be used in both C and C++ programs.
2. Using OS/2 exception handlers. The VisualAge C++ library provides a C-language OS/2 exception handler, `_Exception`, to map OS/2 exceptions to C signals and signal handlers. You can also create and use your own exception handlers.
3. Using C++ exception handling constructs. These constructs belong to the C++ language definition and can only be used in C++ code. C++ exception handling is described in detail in the *Language Reference*.

This chapter describes how to use signal handlers and OS/2 exception handlers alone and in combination. Where appropriate, the interaction between C++ exception handling and the handling of signals and OS/2 exceptions is also described. Both signal and OS/2 exception handling are implemented in C++ as they are in C.

This chapter is only necessary for the advanced programming of exception handling and not for simply debugging exception handling problems. You should use the debugger to debug exception handling problems as complete notification and stack tracing is available through the debugger. OS/2 exceptions and exception handlers are also described in the Toolkit documentation.

Notes:

1. The terms *signal*, *OS/2 exception*, and *C++ exception* are not interchangeable. A signal exists only within the C and C++ languages. An OS/2 exception is generated by the operating system, and may be used by VisualAge C++ library to generate a signal. A C++ exception exists only within the C++ language. In this chapter, the term *exception* refers to an OS/2 exception unless otherwise specified.
2. VisualAge C++ implements C++ exception handling using the OS/2 exception handling facility.

Using C++ and OS/2 Exception Handling in the Same Program

You can make use of C++ exception handling facilities and the OS/2 exception handling facilities in the same program. In fact, VisualAge C++ implements the C++ exception handling facilities using the OS/2 exception handling.

Note: If you use OS/2 exception handling in a program that also uses C++ exception handling, you must not have an OS/2 exception handler that has default behaviour for an unidentified exception. You should always avoid using such an exception handler, but you should be particularly careful to avoid it in programs that use C++ exception handling because the results can be unpredictable.

Handling Signals

Signals are C and C++ language constructs provided for error handling. A signal is a condition reported as a result of an error in program execution. It may also be caused by deliberate programmer action. With the VisualAge C++ product, operating system exceptions are mapped to signals for you. VisualAge C++ product provides a number of different symbols to differentiate between error conditions. The signal constants are defined in the **<signal.h>** header file.

C provides two functions that deal with signal handling in the runtime environment: `raise` and `signal`. Signals can be reported by an explicit call to `raise`, but are generally reported as a result of a machine interrupt (for example, division by zero), of a user action (for example, pressing Ctrl-C or Ctrl-Break), or of an operating system exception.

Use the `signal` function to specify how to handle a particular signal. For each signal, you can specify one of 3 types of handlers:

1. SIG_DFL

   Use the VisualAge C++ default handling. For most signals, the default action is to terminate the process with an error message. See Figure 17 on page 219 for a list of signals and the default action for each. If the `/Tx+` option is specified, the default action can be accompanied by a dump of the machine state to file handle 2, which is usually associated with `stderr`. Note that you can change the destination of the machine-state dump and other messages using the `_set_crt_msg_handle` function, which is described in the *C Library Reference*.

2. SIG_IGN

   Ignore the condition and continue running the program. Some signals cannot be ignored, such as division by zero. If you specify SIG_IGN for one of these signals, the VisualAge C++ library will treat the signal as if SIG_DFL was specified.

3. Your own signal handler function

   Call the function you specify. It can be any function, and can call any library function. Note that when the signal is reported and your function is called, signal handling is reset to SIG_DFL to prevent recursion should the same signal be reported from your function.

The initial setting for all signals is `SIG_DFL`, the default action.

📖 The `signal` and `raise` functions are described in more detail in the *C Library Reference*.

Default Handling of Signals

The runtime environment will perform default handling of a given signal unless a specific signal handler is established or the signal is disabled (set to SIG_IGN). You can also set or reset default handling by coding:

```
signal(sig, SIG_DFL);
```

The default handling depends upon the signal that is being handled. For most signals, the default is to pass the signal to the next exception handler in the chain (the chaining of exception handlers is described in “Registering an OS/2 Exception Handler” on page 240).

Unless you have set up your own exception handler, as described in “Creating Your Own OS/2 Exception Handler” on page 232, the default OS/2 exception handler receives the signal and performs the default action, which is to terminate the program and return an exit code. The exit code indicates:

1. The reason for the program termination. 📖 For the possible values and meanings of the termination code, see `DosExecPgm` in the *Control Program Guide and Reference*.
2. The return code from `DosExit`. 📖 For the `DosExit` return codes, see the *Control Program Guide and Reference*.

The following table lists the C signals that VisualAge C++ runtime library supports, the source of the signal, and the default handling performed by the library.

Figure 17 (Page 1 of 2). Default Handling of Signals

Signal	Source	Default Action
`SIGABRT`	Abnormal termination signal sent by the `abort` function	Terminate the program with exit code 3.

Default Signal Handling

Figure 17 (Page 2 of 2). Default Handling of Signals

Signal	Source	Default Action
`SIGBREAK`	Ctrl-Break signal	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGFPE`	Floating-point exceptions that are not masked[3], such as overflow, division by zero, integer math exceptions, and operations that are not valid	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGILL`	Disallowed instruction	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGINT`	Ctrl-C signal	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGSEGV`	Attempt to access a memory address that is not valid	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGTERM`	Program termination signal sent by the user or operating system	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
`SIGUSR1`	User-defined signal	Ignored.
`SIGUSR2`	User-defined signal	Ignored.
`SIGUSR3`	User-defined signal	Ignored.

3 For more information on masking floating-point exceptions, see“Handling Floating-Point Exceptions” on page 248 .

Establishing a Signal Handler

You can establish or register your own signal handler with a call to the `signal` function:

```
signal(sig, sig_handler);
```

where *`sig_handler`* is the address of your signal handling function. The signal handler is a C function that takes a single integer argument (or two arguments for SIGFPE), and may have either **_System** or **_Optlink** linkage.

A signal handler for a particular signal remains established until one of the following occurs:

- A different handler is established for the same signal.
- The signal is explicitly reset to the system default with the function call `signal(`*`sig`*`, SIG_DFL)`.
- The signal is reported. When your signal handler is called, the handling for that signal is reset to the default as if the function call `signal(sig_num, SIG_DFL)` were explicitly made immediately before the signal handler call.

Note: A signal handler can also become deregistered if the load module where the signal handler resides is deleted using the `_freemod` function. In this situation, when the signal is raised, an OS/2 exception occurs and the behavior is undefined.

Writing a Signal Handler Function

A signal handler function may call any non-critical C library functions. (For a list of critical functions, see Figure 20 on page 231.) Your signal handler may handle the signal in any of the following ways:

1. Calling `exit` or `abort` to terminate the process.
2. Calling `_endthread` to terminate the current thread of a multithread program. The process continues to run without the thread. You must ensure that the loss of the thread does not affect the process. Note that calling `_endthread` for thread 1 of your process is the same as calling `exit`.
3. Calling `longjmp` to go back to an earlier point in the current thread where you called `setjmp`. When you call `setjmp`, it saves the state of the thread at the time of the call. When you call `longjmp` at a later time, the thread is reset to the state saved by `setjmp`, and starts running again at the place where the call to `setjmp` was made.
4. Returning from the function to restart the thread as though the signal has not occurred. If this is not possible, the VisualAge C++ library terminates your process.

Signal Handling Example

Example of a C Signal Handler

The following code gives a simple example of a signal handler function for a single-thread program. In the example, the function `chkptr` checks a given number of bytes in an area of storage and returns the number of bytes that you can access. The flow of the function's execution is described after the code.

```
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <os2.h>

static void mysig(int sig);         /* signal handler prototype */
static jmp_buf jbuf;                /* buffer for machine state */

int chkptr(void * ptr, int size)
{
    void (* oldsig)(int);           /* where to save the old signal handler */
    volatile char c;                /* volatile to ensure access occurs */
    int valid = 0;                  /* count of valid bytes */
    char * p = ptr;

    oldsig = signal(SIGSEGV,mysig); /* set the signal handler */          1

    if (!setjmp(jbuf))              /* provide a point for the      */    2
    {                               /*  signal handler to return to */

       while (size--)                                              ←┐
       {
          c = *p++;                 /* check the storage and */          3
          valid++;                  /*  increase the counter */
       }                                                           ←┘
    }

    signal(SIGSEGV,oldsig);         /* reset the signal handler */       5
    return valid;                   /* return number of valid bytes */   6
}
```

Figure 18 (Part 1 of 2). Example Illustrating a Signal Handler

```
static void mysig(int sig)
{
    UCHAR FileData[100];
    ULONG Wrote;

    strcpy(FileData, "Signal Occurred.\n\r");
    DosWrite(2, (PVOID)FileData, strlen(FileData), &Wrote);          4
    longjmp(jbuf,1);       /* return to the point of the setjmp call */

}
```

Figure 18 (Part 2 of 2). Example Illustrating a Signal Handler

1 The program registers the signal handler `mysig` and saves the original handler in `oldsig` so that it can be reset at a later time.

2 The call to `setjmp` saves the state of the thread in `jbuf`. When you call `setjmp` directly, it returns `0`, so the code within the `if` statement is run.

3 The loop reads in and checks each byte of the buffer, incrementing `valid` for each byte successfully copied to `c`.

Assuming that not all of the buffer space is available, at some point in the loop `p` points to a storage location the process cannot access. An OS/2 exception is generated and translated by the VisualAge C++ library to the SIGSEGV signal. The library then resets the signal handler for SIGSEGV to SIG_DFL and calls the signal handler registered for SIGSEGV (`mysig`).

4 The `mysig` function prints an error message and uses `longjmp` to return to the place of the `setjmp` call in `chkptr`.

Note: `mysig` does not reset the signal handler for SIGSEGV, because that signal is not intended to occur again. In some cases, you may want to reset signal handling before the signal handler function ends.

5 Because `setjmp` returns a nonzero value when it is called through `longjmp`, the `if` condition is now false and execution falls through to this line. The signal handling for SIGSEGV is reset to whatever it was when `chkptr` was entered.

6 The function returns the number of valid bytes in the buffer.

As the preceding example shows, your program can recover from a signal and continue to run successfully.

Signal Handling in Multithread Programs

Each thread has its own set of signals, and signal handlers are registered independently on each thread. If you establish a signal handler or raise a signal on one thread, you do not affect any other thread. For example, if thread 1 calls `signal` as follows:

```
signal(SIGFPE, handlerfunc);
```

then the handler `handlerfunc` is registered for thread 1 only. All other threads are handled using the defaults.

When a thread starts, all of its signal handlers are set to SIG_DFL. If you want any other signal handling for that thread, you must explicitly register it using `signal`.

A signal is always handled on the thread that generated it, except for SIGBREAK, SIGINT, and SIGTERM. These three signals are handled on the thread that generated them only if they were raised using the `raise` function. If they were raised by an exception, they will be handled on thread 1. Thus to establish a signal handler for them, you must call `signal` in thread 1.

When you call the `raise` function, the handler for the signal you raise must be established on the thread where the call was made.

Note: You can use `raise` to signal your own conditions using the signals SIGUSR1, SIGUSR2, and SIGUSR3. You can also use this function to generate signals to test your signal handlers.

Signal Handling Considerations

When you use signal handlers, keep the following points in mind:

- You can register anything as a signal handler. It is up to you to make sure that you are registering a valid function.
- If your signal handler resides in a DLL, ensure that you change the signal handler when you unload the DLL. If you unload your DLL without changing the signal handler, no warnings or error messages are generated. When your signal handler gets called, your program will probably terminate. If another DLL has been loaded in the same address range, your program may continue but with undefined results.
- Your signal handler should not assume that SIGSEGV always implies an invalid data pointer. It can also occur, for example, if an address pointer goes outside of your code segment.

- The SIGILL signal is not guaranteed to occur when you call an invalid function using a pointer. If the pointer points to a valid instruction stream, SIGILL is not raised.
- When you use `longjmp` to leave a signal handler, ensure that the buffer you are jumping to was created by the thread that you are in. Do not call `setjmp` from one thread and `longjmp` from another. The VisualAge C++ library terminates a process where such a call is made.
- If you use console I/O functions, including `gets` and `scanf`, and a SIGINT, SIGBREAK, or SIGTERM signal occurs, the signal is reported **after** the library function returns. Because your signal handler can call any non-critical library function, one of these functions could be reentered. To protect the internal data structures, some library code is placed in "must complete" sections. When a signal occurs, the library waits until the "must complete" section ends before it reports the signal.

  Note: You can use the OS/2 APIs `DosEnterMustComplete` and `DosExitMustComplete` to create your own "must complete" sections of code. See the *Control Program Guide and Reference* for more information on these APIs.
- Variables referenced by both the signal handler and by other code should be given the attribute `volatile` to ensure they are always updated when they are referenced. Because of the way the compiler optimizes code, the following example may not work as intended when compiled with the `/O+` option:

```
void sig_handler(int);
static int stepnum;

int main(void)
{
    stepnum = 0;
    signal(SIGSEGV, sig_handler);

  ⋮
    stepnum = 1;    1

  ⋮
    stepnum = 2;    2
}

void sig_handler(int x)
{
    UCHAR FileData[100];
    ULONG Wrote;

    strcpy(FileData, "Error at Step %d\n\r");
    DosWrite(2, (PVOID)FileData, strlen(FileData), &Wrote, stepnum);
}
```

When using optimization, the compiler may not immediately store the value 1 for the variable stepnum. It may never store the value 1, and store only the value 2. If a signal occurs between statement **1** and statement **2**, the value of stepnum passed to sig_handler may not be correct.

Declaring stepnum as volatile indicates to the compiler that references to this variable have side effects. Changes to the value of stepnum are then stored immediately.

- **C++ Consideration:** When you use longjmp to recover from a signal in a C++ program, automatic destructors are not called for objects placed on the stack between the longjmp call and the corresponding setjmp call. Because the ANSI draft of the C++ language does not specify the behavior of a throw statement in a signal handler, the most portable way to ensure the appropriate destructors are called is to add statements to the setjmp location that will do a throw if necessary.

Handling OS/2 Exceptions

An OS/2 exception is generated by the operating system to report an abnormal condition. OS/2 exceptions are grouped into two categories:

1. Asynchronous exceptions, which are caused by actions outside of your current thread. There are only two:
   - XCPT_SIGNAL, caused by a keyboard signal (Ctrl-C, Ctrl-Break) or the process termination exception. This exception can only occur on thread 1 of your process.
   - XCPT_ASYNC_PROCESS_TERMINATE, caused by one of your threads terminating the entire process. This exception can occur on any thread.
2. Synchronous exceptions, which are caused by code in the thread that receives the exception. All other OS/2 exceptions fall into this category.

Just as you use signal handlers to handle signals, use exception handlers to handle OS/2 exceptions. Although exception handling offers additional function, because signal handling is simpler you may want to use both.

VisualAge C++ Default OS/2 Exception Handling

The VisualAge C++ library provides its own default exception handling functions: `_Lib_excpt`, for OS/2 exceptions occurring in library functions, and `_Exception`, for all other OS/2 exceptions. You can use these exception handlers or create your own as described in "Creating Your Own OS/2 Exception Handler" on page 232.

The function `_Exception` is the C language exception handler. It is declared as:

```
#include <os2.h>

unsigned long _System _Exception(EXCEPTIONREPORTRECORD * report_rec,
                                 EXCEPTIONREGISTRATIONRECORD * reg_rec,
                                 CONTEXTRECORD * exc,
                                 void * dummy);
```

This exception handler is registered by the VisualAge C++ compiler for every thread or process that is started by `_beginthread`, unless **`#pragma handler`** is specified for the function. The function `_Exception` maps recognized OS/2 exceptions to C signals, which can then be passed by the runtime library to the appropriate signal handlers.

VisualAge C++ Default OS/2 Exception Handling

Figure 19 shows which types of OS/2 exception are recognized by `_Exception`, the names of the exceptions, and the C signals to which each exception type is mapped. **These are the only OS/2 exceptions handled by** `_Exception`. The **Continuable** column indicates whether the program will continue if the corresponding signal handler is `SIG_IGN` or if a user-defined signal handler returns. If "No" is indicated, the program can only be continued if you provide a signal handler that uses `longjmp` to jump to another part of the program.

If the signal handler value is set to `SIG_DFL`, the default action taken for each of these exceptions is to terminate the program with an exit code of 99.

Figure 19 (Page 1 of 2). Mapping Between Exceptions and C Signals

OS/2 Exception	**C Signal**	**Continuable?**
Divide by zero `XCPT_INTEGER_DIVIDE_BY_ZERO`	`SIGFPE`	No

NPX387 error
`XCPT_FLOAT_DENORMAL_OPERAND`
`XCPT_FLOAT_DIVIDE_BY_ZERO`
`XCPT_FLOAT_INEXACT_RESULT`
`XCPT_FLOAT_INVALID_OPERATION`
`XCPT_FLOAT_OVERFLOW`
`XCPT_FLOAT_STACK_CHECK`
`XCPT_FLOAT_UNDERFLOW`
	`SIGFPE`	No; except for `XCPT_FLOAT_INEXACT_RESULT`

Overflow occurred `XCPT_INTEGER_OVERFLOW`	`SIGFPE`	Yes; resets the overflow flag
Bound opcode failed `XCPT_ARRAY_BOUNDS_EXCEEDED`	`SIGFPE`	No

Opcode not valid
`XCPT_ILLEGAL_INSTRUCTION`
`XCPT_INVALID_LOCK_SEQUENCE`
`XCPT_PRIVILEGED_INSTRUCTION`
	`SIGILL`	No

General Protection fault
`XCPT_ACCESS_VIOLATION`
`XCPT_DATATYPE_MISALIGNMENT`
	`SIGSEGV`	No

Ctrl-Break `XCPT_SIGNAL` (`XCPT_SIGNAL_BREAK`)	`SIGBREAK`	Yes
Ctrl-C `XCPT_SIGNAL` (`XCPT_SIGNAL_INTR`)	`SIGINT`	Yes

Figure 19 (Page 2 of 2). Mapping Between Exceptions and C Signals

OS/2 Exception	C Signal	Continuable?
End process `XCPT_SIGNAL (XCPT_SIGNAL_KILLPROC)`	`SIGTERM`	Yes

Note: The Integer Overflow and Bound opcode exceptions will never be caused by code generated by VisualAge C++ compiler.

The following OS/2 exceptions are also recognized, but have no corresponding C signal. If one of these OS/2 exceptions occurs, it is passed to the next available exception handler, or if none is registered, it is passed to the operating system:

OS/2 Exception	Continuable?
Out of stack exception `XCPT_GUARD_PAGE_VIOLATION`	Yes
Synchronous process termination `XCPT_PROCESS_TERMINATE`	No
Asynchronous process termination `XCPT_ASYNC_PROCESS_TERMINATE`	No
Unwind target not valid `XCPT_INVALID_UNWIND_TARGET`	No

An out-of-stack exception occurs when the guard page of the stack is accessed. When the operating system encounters this exception, it automatically allocates a new guard page and the exception is continued. A nested-unable-to-grow-stack exception occurs when a guard page violation cannot be processed because there is insufficient stack space. Stack probes are also required to make automatic stack growth work properly.

For more information on guard page allocation and automatic stack growth, see the *User's Guide*.

OS/2 Exception Handling in Library Functions

There are two classes of library functions that require special exception handling: math functions and critical functions.

OS/2 exceptions occurring in all other library functions are treated as though they occurred in regular user code.

Library Exception Handling

Math Functions

Before _Exception converts an OS/2 exception to a C signal, it first calls the VisualAge C++ library exception handler, _Lib_excpt. The _Lib_excpt function determines if the exception occurred in a math library function. The _Lib_excpt function is declared as follows:

```
#include <os2.h>

unsigned long _System _Lib_excpt(EXCEPTIONREPORTRECORD * report_rec,
                                 EXCEPTIONREGISTRATIONRECORD * reg_rec,
                                 CONTEXTRECORD * ecx,
                                 void * dummy);
```

If the exception does occur in a math function and it is a floating-point error, _Lib_excpt handles the exception and returns XCPT_CONTINUE_EXECUTION to the operating system to indicate the exception has been handled. Any signal handler function you may have established will **not** be called.

Important: If you are creating your own exception handler, it should first call _Lib_excpt to ensure that the exception did not occur in a library function.

If the cause of the OS/2 exception was not a floating-point error, the exception is returned to _Exception. The _Exception function then converts the OS/2 exception to the corresponding C signal and performs one of the following actions:

1. Terminates the process. If /Tx+ was specified, _Exception performs a machine-state dump to file handle 2, unless the exception was SIGBREAK, SIGINT, or SIGTERM, in which case the machine state is not meaningful.
2. Handles the exception and returns XCPT_CONTINUE_EXECUTION to the operating system.
3. Calls the signal handler function provided by you for that signal. A return from the signal handler results in either the return of XCPT_CONTINUE_EXECUTION to the operating system or the termination of the process as in the first action above.

For more information about exception-handling return codes, refer to the *Control Program Guide and Reference*.

Critical Functions

All nonreentrant functions are classified as critical functions. Most I/O and allocation functions, and those that begin or end threads or processes, fall in this class. The critical functions are:

Figure 20. Critical Functions

atexit	execv	freopen	putenv	tempnam
calloc	execve	fscanf	puts	_tfree
_cgets	execvp	fseek	raise	_theapmin
clearerr	_execvpe	fsetpos	realloc	_tmalloc
_cprintf	exit	ftell	remove	tmpfile
_cputs	fclose	fwrite	rename	tmpnam
_cscanf	_fcloseall	_getch	rewind	_trealloc
_debug_calloc	fdopen	_getche	_rmtmp	_uaddmem
_debug_free	feof	getenv	scanf	_ucalloc
_debug_heapmin	ferror	gets	setlocale	_ucreate
_debug_malloc	fflush	_heap_check	setvbuf	_udefault
_debug_realloc	fgetc	_heapchk	signal	_udestroy
_debug_ucalloc	fgetpos	_heapmin	_spawnl	_udump_allocated
_debug_uheapmin	fgets	_heapset	_spawnle	_dump_allocated_delta
_debug_umalloc	fileno	_heap_walk	_spawnlp	_udump_allocated_delta
_dump_allocated	_flushall	_interupt	_spawnlpe	ungetc
_endthread	fopen	_kbhit	_spawnv	_ungetch
_Exception	fprintf	_Lib_excpt	_spawnve	_uheapchk
execl	fputc	malloc	_spawnvp	_uheapmin
execle	fputs	_onexit	_spawnvpe	_uheapset
execlp	fread	printf	system	_uheap_walk
_execlpe	free	_putch	_tcalloc	_umalloc
				vfprintf
				vprintf

OS/2 exceptions in critical functions generally occur only if your program passes a pointer that is not valid to a library function, or if your program overwrites the library's data areas. Because calling a signal handler to handle an OS/2 exception from one of these functions can have unexpected results, a special exception handler is provided for critical functions. **You cannot override this exception handler**.

If the OS/2 exception is synchronous (SIGFPE, SIGILL, or SIGSEGV), the default action is taken, which is to pass the exception on to the next registered exception handler. Any exception handler you may have registered will **not** be called, and will receive only the termination exception.

If the OS/2 exception is asynchronous, it is deferred until the library function has finished. The exception is then passed to _Exception, which converts the exception to the corresponding C signal and performs the appropriate action.

Note: If you use console I/O functions (for example, gets) and a SIGINT, SIGBREAK, or SIGTERM signal occurs, the signal is deferred until the function returns, for example, after all data for the keyboard function has been entered. To avoid this side effect, use a noncritical function like `read` or the OS/2 API `DosRead` to read data from the keyboard.

Creating Your Own OS/2 Exception Handler

You can use OS/2 APIs and the information provided in the **<bsexcpt.h>** header file found in the `\IBMCPP\INCLUDE\os2` directory to create your own exception handlers to use alone or with the two provided handler functions. Exception handlers can be complex to write and difficult to debug, but creating your own offers you two advantages:

1. You receive more information about the error condition.
2. You can intercept any OS/2 exception. The VisualAge C++ library passes some exceptions back to the operating system because there is no C semantic for handling them.

Prototype of an OS/2 Exception Handler

The prototype for all exception handlers is:

```
#define INCL_BASE
#include <os2.h>

APIRET APIENTRY MyExceptHandler(EXCEPTIONREPORTRECORD *,
                                EXCEPTIONREGISTRATIONRECORD *,
                                CONTEXTRECORD *,
                                PVOID  dummy);
```

where:

`APIRET` Specifies the return type of the function. If you return from your exception handler, you must return one of the following two values:

1. `XCPT_CONTINUE_SEARCH` indicates that the exception has not been handled and tells the operating system to pass the exception to the next exception handler.
2. `XCPT_CONTINUE_EXECUTION` indicates that the exception condition has been corrected and tells the operating system to resume running the application using the information in the `CONTEXTRECORD`.

`APIENTRY`
Defines the function linkage. The OS/2 header files found in the `ibmcpp\include\os2` directory header files define `APIENTRY` as **_System** linkage. Use the `APIENTRY` keyword rather than specifying the linkage type yourself. Note that your exception handler must be an external function; it cannot be `static`.

EXCEPTIONREPORTRECORD *
: Points to a structure that contains high-level information about the exception.

EXCEPTIONREGISTRATIONRECORD *
: Points to the record that registered the exception handler. The address of the record is always on the stack.

CONTEXTRECORD *
: Points to a structure that contains information about the state of the thread at the time of the exception, including the register contents and the state of the floating-point unit and flags. When an exception handler returns XCPT_CONTINUE_EXECUTION, the machine state is reloaded from this structure. You should only modify the contents of this structure if you are sure your exception handler will return XCPT_CONTINUE_EXECUTION.

PVOID
: Is a pointer to void that you must pass back unchanged to the operating system.

The exception handling structures are defined in the **<bsexcpt.h>** header file found in the ibmcpp\include\os2 directory.

Processing Exception Information

When an exception occurs, the operating system provides a considerable amount of information. Of this, the information contained in the EXCEPTIONREPORTRECORD structure can be quite useful.

The EXCEPTIONREPORTRECORD is defined as:

```
struct _EXCEPTIONREPORTRECORD
   {
   ULONG   ExceptionNum;
   ULONG   fHandlerFlags;
   struct  _EXCEPTIONREPORTRECORD *NestedERR;
   PVOID   ExceptionAddress;
   ULONG   cParameters;
   ULONG   ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS];
   };
```

The structure fields provide the following information:

`ExceptionNum`
: The exception number. There are several exceptions that you will only encounter by using an OS/2 exception handler because the VisualAge C++ default handler passes them to the operating system to handle. They are:

  `XCPT_PROCESS_TERMINATE`
  : Indicates that the current thread has called `DosExit`, and the process is about to end. Until your exception handler ends, the thread continues as though `DosExit` had not been called.

  `XCPT_ASYNC_PROCESS_TERMINATE`
  : Indicates that some other thread in the process has called `DosExit` and that your current thread is about to end also. You can decide to continue running the current thread and return the exception as handled.

  `XCPT_ACCESS_VIOLATION`
  : Indicates an invalid attempt was made to access memory (similar to the SIGSEGV signal). When this exception occurs, the `ExceptionInfo` field provides the address that generated the exception and the type of access that was attempted (read or write).

  `XCPT_GUARD_PAGE_VIOLATION`
  : Indicates that the current thread tried to access a memory page marked as a guard page. Usually it means that your application has accessed a guard page on the stack. In most cases, you will probably pass the exception to the operating system, which will allocate another 4K of committed memory for your thread and a new guard page. The operating system requires about 1.5K to place the information about the exception on the stack and then call the exception handler. If you know you are running out of stack space, you may want to end your process.

  `XCPT_UNABLE_TO_GROW_STACK`
  : Indicates that the operating system tried to move your guard page, but no memory remained on the stack. If you suppressed stack probe generation when you compiled (with the `/Gs+` option), there may not be enough stack for you to even receive the exception, in which case your process terminates with an operating system trap.

  You can also use the `DosRaiseException` API to create and raise your own exceptions that you can then handle with your own exception handler.

fHandlerFlags
: This field indicates how the exception occurred and what you can do to handle it. It includes the following bits:

  EH_NONCONTINUABLE
  : You cannot continue running the thread once you leave the exception handler. If you try to return XCPT_CONTINUE_EXECUTION, an error is generated. You cannot reset the bit. However, you can intentionally set the bit to make an exception noncontinuable.

  EH_UNWINDING
  : A longjmp has been done over this exception handler and the handler is to be deregistered. If your function uses a mutex semaphore (described in the Toolkit documentation), you should release it when you receive this exception.

  EH_EXIT_UNWIND
  : A DosExit call has been made and the exception has been passed back to the operating system. This exception gives you an opportunity to do something before your exception handler is deregistered.

  EH_NESTED_CALL
  : An exception occurred while another exception was being handled. This situation should be handled carefully: each exception requires about 1.5K of stack, so nesting exceptions too deep can cause you to run out of stack.

_EXCEPTIONREPORTRECORD *NestedERR
: If a nested exception occurs, the information about the exception is found in this structure.

ExceptionAddress
: This field contains the instruction address where the exception occurred. Typically, you cannot determine at run time which function caused the problem.

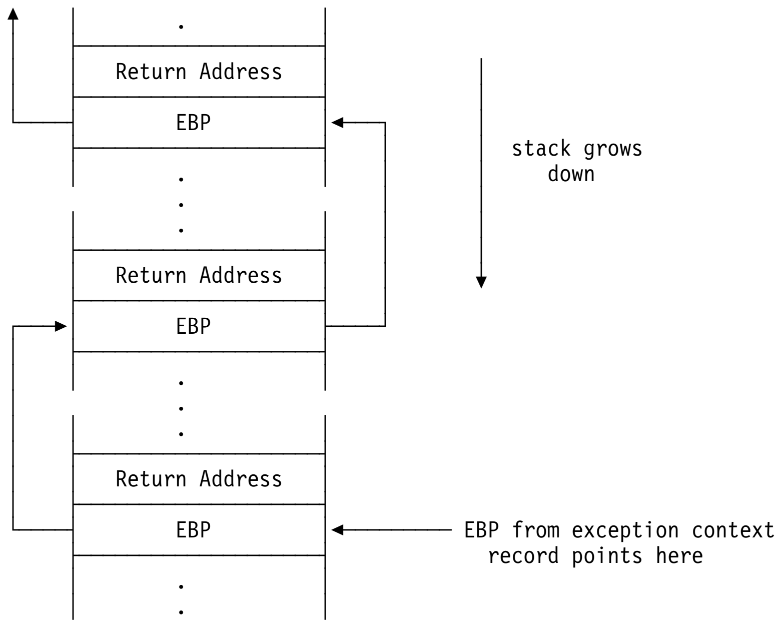
ExceptionInfo
: For some exceptions, this field may contain additional information. For example, if XCPT_ACCESS_VIOLATION occurs, it contains the address at which the memory access failed.

cParameters
: This field contains the number of bytes of information.

The CONTEXTRECORD structure contains information about the machine state of the thread. It is generally of limited use to a high-level programmer because to continue a process after a synchronous exception, you would need to modify the CONTEXTRECORD, and it is extremely difficult to ensure the exception handler code is correct for all possible conditions. You should modify the CONTEXTRECORD only if you have no other alternative to correct your program.

You **can** use the CONTEXTRECORD to trace the stack and produce useful debugging information. Because the VisualAge C++ and operating system calling conventions preserve some registers across calls, you cannot reconstruct the registers by traversing the stack to recover from the exception.

The 32-bit stack usually looks like the following:



Notes:

1. If your code is optimized, you may not be able to trace the EBP chain.
2. If the stack is damaged, you may not be able to trace the EBP chain correctly.
3. You cannot trace over 16-bit calls.
4. You cannot trace over calls to inlined functions.

Example of Exception Handling

The following example shows a program similar to the one used for the signal handling example on page 222. In this example, an exception handler is used instead of a signal handler to detect access to memory that is not valid.

```
#define  INCL_DOS
#define  INCL_NOPMAPI
#include <os2.h>
#include <stdlib.h>
#include <setjmp.h>
#include <stdio.h>
#include <stddef.h>       /* for _threadid */

void * tss_array[100];     /* array for 100 thread-specific pointers */

APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD *,
                                   EXCEPTIONREGISTRATIONRECORD *,
                                   CONTEXTRECORD *,
                                   PVOID);
#pragma map(_Exception,"MyExceptionHandler")
#pragma handler(chkptr)

int chkptr(void * ptr, int size)
{
    volatile char c;       /* volatile to insure access occurs */
    int valid = 0;         /* count of valid bytes */
    char * p = ptr;        /* to satisfy the type checking for p++ */
    jmp_buf jbuf;          /* put the jump buffer in automatic storage */
                           /*   so it is unique to this thread         */

    PTIB ptib;             /* to get the TIB pointer */
    PPIB ppib;
    unsigned int tid = *_threadid;   /* get the thread id */
    UCHAR FileData [100]
    ULONG Wrote;

    /* create a thread specific jmp_buf */
    tss_array[tid] = (void *) jbuf;
```

Figure 21 (Part 1 of 3). Example Illustrating an Exception Handler

```
    if (!setjmp(jbuf)) {   /* provide a point to return to */

       while (size--)      /* scan the storage */
       {
          c = *p++;
          valid++;
       }
    }

    return valid;                    /* return number of valid bytes */
}

/* the exception handler itself */

APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD * report_rec,
                                   EXCEPTIONREGISTRATIONRECORD * register_rec,
                                   CONTEXTRECORD * context_rec,
                                   PVOID dummy)
{
   unsigned int tid = *_threadid;   /* get the thread id */

   /* check the exception flags */         1
   if (EH_EXIT_UNWIND & report_rec->fHandlerFlags) /* exiting */
      return XCPT_CONTINUE_SEARCH;

   if (EH_UNWINDING & report_rec->fHandlerFlags)   /* unwinding */
      return XCPT_CONTINUE_SEARCH;

   if (EH_NESTED_CALL & report_rec->fHandlerFlags) /* nested exceptions */
      return XCPT_CONTINUE_SEARCH;

   /* determine what the exception is */   2
   if (report_rec->ExceptionNum == XCPT_ACCESS_VIOLATION) {
      /* this is the one that is expected */
```

Figure 21 (Part 2 of 3). Example Illustrating an Exception Handler

```
    UCHAR FileData[100];
    ULONG Wrote;

    strcpy(FileData, "Detected invalid storage address %d\n\r");
    DosWrite(2, (PVOID)FileData, strlen(FileData), &Wrote, stepnum);
      longjmp((int *)tss_array[tid],1);   /* return to the point of the */
                                          /* setjmp call without        */
                                          /* restarting the while loop  */
   } /* endif */
                                          3
   return XCPT_CONTINUE_SEARCH;      /* if it is a different exception */
}
```

Figure 21 (Part 3 of 3). Example Illustrating an Exception Handler

1 The first thing an exception handler should do is check the exception flags. If `EH_EXIT_UNWIND` is set, meaning the thread is ending, the handler tells the operating system to pass the exception to the next exception handler. It does the same if the `EH_UNWINDING` flag is set, the flag that indicates this exception handler is being removed.

The `EH_NESTED_CALL` flag indicates whether the exception occurred within an exception handler. If the handler does not check this flag, recursive exceptions could occur until there is no stack remaining.

2 The handler checks the exception number. In general, you should check for only the exceptions that you expect to encounter so any addition of new exception numbers does not affect your code. Assuming the exception is `XCPT_ACCESS_VIOLATION`, the exception handler prints a message and calls `longjmp` to return to the `chkptr` function.

3 If the exception is not the expected one, the handler tells the operating system to pass it to the next exception handler.

Important: Return XCPT_CONTINUE_EXECUTION from an exception handler **only** if you know that the thread can continue to run because either:

1. The exception is asynchronous and can be restarted.
2. You have changed the thread state so that the thread can continue.

If you return XCPT_CONTINUE_EXECUTION when neither of these conditions is true, you could generate a new exception each time your exception handler ends, eventually causing your process to lock.

Registering an OS/2 Exception Handler

The VisualAge C++ compiler automatically registers and deregisters the _Exception handler for each thread or process so the _Exception is the first exception handler to be called when an exception occurs. To explicitly register _Exception for a function, use the **#pragma handler** directive before the function definition. This directive generates the code to register the exception handler before the function runs. Code to remove the exception handler when the function ends is also generated.

The format of the directive is:

```
#pragma handler(function)
```

where *function* is the name of the function for which the exception handler is to be registered.

Note: If you use DosCreateThread to create a new thread, you **must** use **#pragma handler** to register the VisualAge C++ exception handler for the function that the new thread will run.

You can register your own exception handler in place of _Exception using these directives:

```
#pragma map(_Exception, "MyExceptHandler")
#pragma handler(myfunc)
```

The **#pragma map** directive tells the compiler that all references to the name _Exception are to be converted to MyExceptHandler. The **#pragma handler** directive would normally register the exception handler _Exception for the function myfunc, but because of the name mapping, MyExceptHandler is actually registered. The compiler also generates code to deregister MyExceptHandler when myfunc returns.

Registering an OS/2 Exception Handler

If you use the method described above, you can have only one exception handler per module. You may need to place functions in separate modules to get the exception handling you want. The handler is registered on function entry and deregistered on exit; you cannot register the handler over only part of a function. For more flexibility, you can use OS/2 APIs to register your exception handler.

The operating system finds exception handlers by following a chain rooted in the thread information block (TIB). When you register an exception handler, you place the address of the handler and the chain pointer from the TIB in an EXCEPTIONREGISTRATIONRECORD structure, and then update the TIB to point to the new EXCEPTIONREGISTRATIONRECORD.

When you use **#pragma handler**, the EXCEPTIONREGISTRATIONRECORD is generated and attached to the chain for you. You can register your own records using the DosSetExceptionHandler and DosUnsetExceptionHandler APIs, as shown in the following example:

```
#define INCL_BASE
#include <os2.h>

/* the prototype for the exception handler */
APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD *,
                                   EXCEPTIONREGISTRATIONRECORD *,
                                   CONTEXTRECORD *,
                                   PVOID);
int myfunction(...)
{
   EXCEPTIONREGISTRATIONRECORD err = { NULL,MyExceptionHandler };

   DosSetExceptionHandler(&err);  /* register */
   .
   .
   .
   DosUnsetExceptionHandler(&err);  /* deregister */
}
```

Using the OS/2 APIs provides more flexibility than using **#pragma handler**. You can register the exception handler over only a part of the function if you want. You can also register more than one exception handler for a function. When you use DosSetExceptionHandler to register your handler, you can also make the EXCEPTIONREGISTRATIONRECORD part of a larger structure and then access the information in that structure from inside the exception handler.

Registering an OS/2 Exception Handler

You **must** deregister the exception handler before the function ends. If you do not, the next exception that occurs on the thread can have unexpected and undefined results. When you use **`pragma handler`**, the exception handler is automatically deregistered for you.

The following diagram shows the TIB chain:

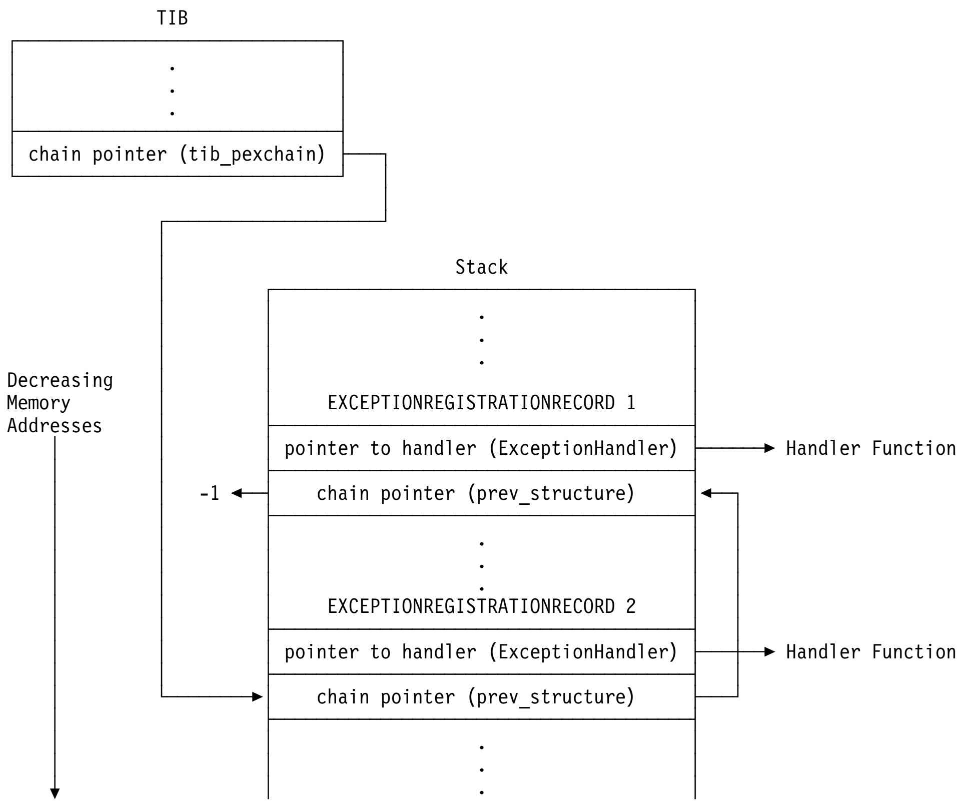

TIB
chain pointer (tib_pexchain)
Stack
Decreasing Memory Addresses
EXCEPTIONREGISTRATIONRECORD 1
pointer to handler (ExceptionHandler)
Handler Function
-1
chain pointer (prev_structure)
EXCEPTIONREGISTRATIONRECORD 2
pointer to handler (ExceptionHandler)
Handler Function
chain pointer (prev_structure)

Figure 22. TIB Chain. Names in parentheses are the names of the fields of the `EXCEPTIONREGISTRATIONRECORD` *structure.*

Each `EXCEPTIONREGISTRATIONRECORD` is chained to the next. When an exception occurs, the operating system begins at the TIB and goes to each `EXCEPTIONREGISTRATIONRECORD` in turn. It calls the exception handler and passes it the exception information. The exception handler either handles the exception or tells the operating system to pass the exception to the next handler in the chain. If the last exception handler in the chain, identified by chain pointer with value -1, does not handle the exception, the operating system takes the default action.

An `EXCEPTIONREGISTRATIONRECORD` must be on the stack, and each record must be at a higher address than the previous one.

Handling Signals and OS/2 Exceptions in DLLs

Handling signals and OS/2 exceptions in DLLs is no different than handling signals in executable files, providing all your DLLs and the executable files that use them are created using the VisualAge C++ compiler, and only one VisualAge C++ library environment exists for your entire application (your executable module and all DLLs).

The library environment is a section of information associated with and statically linked to the VisualAge C++ library itself. You can be sure your program has only one library environment if:

1. It consists of a single executable module. By definition, a single module has only one copy of the VisualAge C++ library environment regardless of whether it links to the library statically or dynamically.
2. Your executable module dynamically links to a single DLL that is statically bound to the VisualAge C++ runtime library and that uses the VisualAge C++ library functions. The executable module then accesses the library functions through the DLL.
3. Your executable modules and DLLs all dynamically link to the VisualAge C++ runtime library.

Note: The licensing agreement does not allow you to ship the VisualAge C++ library DLLs with your application. You can, however, create your own version of the runtime library and dynamically link to it from all of your modules, ensuring that only one copy of the library environment is used by your application. If you call any VisualAge C++ library functions from a user DLL, you must call them all from that DLL. The method of creating your own runtime library is described in "Creating Your Own Runtime Library DLLs" on page 83.

Signal/Exception Handling in DLLs

If more than one of your modules is statically linked to the VisualAge C++ library, your program has more than one library environment. Because there is no communication between these environments, certain operations and functions become restricted:

- Stream I/O. You can pass the file pointer between modules and read to or write from the stream in any module, but you cannot open a stream in one library environment or module and close it in another.
- Memory allocation. You can pass the storage pointer between modules, but you cannot allocate storage in one library environment and free or reallocate it in another.
- `strtok`, `rand`, and `srand` functions. A call to any of these functions in one library environment has no effect on calls made in another environment.
- **errno** and **_doserrno** values. The setting of these variables in one library environment has no effect on their values in another.
- Signal and OS/2 exception handlers. The signal and exception handlers for a library environment have no effect on the handlers for another environment.

In general, it is easier to use only one library environment, but not always possible. For example, if you are building a DLL that will be called by a number of applications, you should assume that there may be multiple library environments and code your DLL accordingly.

The following section describes how to use signal and exception handling when your program has more than one library environment.

Signal and Exception Handling with Multiple Library Environments

When you have multiple library environments, you must treat signal and exception handlers in a slightly different manner than you would with a single library environment. Otherwise, the wrong handler could be called to handle a signal or OS/2 exception.

For example, if you have an executable module and a DLL, each with its own library environment, the `_Exception` exception handler is automatically registered for the executable module when it starts. When the executable module calls a function in the DLL, the thread of execution passes to the DLL. If an OS/2 exception then occurs in the code in the DLL, it is actually handled by the exception handler in the executable module's library environment. Any signal handling set up in the DLL is ignored.

When you have more than one library environment, you must ensure that an OS/2 exception is always handled by the exception handler for the library environment where the exception occurred.

Include **`#pragma handler`** statements in your DLL for every function in the DLL that can be called from another module. This directive ensures the exception handler for the DLL's library environment is correctly registered when the function is called and deregistered when the function returns to the calling module. If functions in your executable module can themselves be called back to from a DLL, include a **`#pragma handler`** statement for each of them also.

Using OS/2 Exception Handlers for Special Situations

Using exception handlers can be especially helpful in the following situations:

- In multithread programs that use OS/2 semaphores. If you acquire a semaphore and then use `longjmp` either explicitly or through a signal handler to move to another place in your program, the semaphore is still owned by your code. Other threads in your program may not be able to obtain ownership of the semaphore.

  If you register an exception handler for the function where the semaphore is requested, the handler can check for the unwind operation that occurs as a result of a `longjmp` call. If it encounters an unwind operation, it can then release the semaphore.

- In system DLLs. Using an exception handler allows you to run process termination routines even if your DLL has global initialization and termination.

  When a process terminates, functions are called in the following order:

  1. Functions registered with the `atexit` or `_onexit` functions.
  2. Exception handlers for termination exceptions.
  3. Functions registered with the `DosExitList` API.
  4. DLL termination routines.

  You can include process termination routines in your exception handler and they will be performed before the DLL termination routines are called.

OS/2 Exception Handling Considerations

All the restrictions for signal handling described on page 224 apply to exception handling as well. There are also a number of additional considerations you should keep in mind when you use exception handling:

- You **must** register an exception handler whenever you change library environments to ensure that exception handling is provided for all C code.
- Ensure that you always deregister your exception handler. If you do not, your process typically ends abnormally. It is very difficult to discover this problem through debugging. If you use **`#pragma handler`**, the handler is automatically deregistered; if you use the OS/2 APIs, you must call `DosUnsetExceptionHandler`.
- If you register your own exception handler, the OS/2 exceptions you handle are not seen by a signal handler. The exceptions you do not handle are passed to the next exception handler. If the next handler is the VisualAge C++ default handler `_Exception`, it converts the exception to a signal and calls the appropriate signal handler.
- If you are using OS/2 semaphores and an exception occurs while your code owns a semaphore, you must ensure that the semaphore is released. You can release the semaphore either by continuing the exception or by explicitly releasing the semaphore in the signal handler.
- Always check the exception flags to determine how the exception occurred. Any exception handler can be unwound by a subsequent handler.
- Keep your exception handler simple and specific. Exception handlers are easier to write and maintain if you limit what they can do. A handler that does everything can be very large and very complicated.
- Check for and handle only the exceptions that you expect to encounter, and provide a default exception handler to handle the unexpected. If the operating system adds new exceptions, or if you create your own, the default handler will handle them.
- If you are using your own exception handler, it receives the exception registration record when an exception occurs, as described in "Registering an OS/2 Exception Handler" on page 240. Do **not** use the return address of the calling function to tell you where to resume execution, because the values of the registers other than EBP (for example, EBX, EBI, and EDI) at the return are generally not available to your exception handler.
- You need approximately 1.5K of stack remaining for the operating system to be able to call your exception handler. If you do not have enough stack left, the operating system terminates your process.

- Neither of the VisualAge C++ default exception handlers are available in the subsystem libraries. Because the subsystem libraries contain no critical or math functions, the `_Lib_excpt` function is not required.

Restricted OS/2 APIs

When you use the VisualAge C++ default exception handlers, certain OS/2 APIs can interfere with exception handling:

`DosCreateThread`
: This API does not automatically register an exception handler for the new thread. Use `_beginthread` instead, or use **`#pragma handler`** before the `DosCreateThread` call to register the handler for the thread.

`DosExit`
: This API does not perform all necessary library termination routines. Instead, use `exit` or `_exit`, `abort`, or `_endthread`, or simply fall out of `main`.

`DosUnwindException`
: This API can unwind or remove the VisualAge C++ exception handlers from the stack. Use `longjmp` instead.

`DosSetSignalExceptionFocus`
: Using this API to remove the signal focus from a VisualAge C++ application may prevent you from receiving SIGINT and SIGBREAK exceptions from the keyboard.

`DosAcknowledgeSignalException`
: This API interferes with the VisualAge C++ handling of signal exceptions. The library exception handler acknowledges signals for you.

`DosEnterMustComplete`
: This API can be used to delay the handling of asynchronous exceptions, including termination exceptions, until a section of code has ended. You must call `DosExitMustComplete` at the end of the section to reenable the exception handling.

`DosEnterCritSec`
: This API prevents execution from switching between threads until a section of code has ended. You must call `DosExitCritSec` at the end of the critical section of code. Use these APIs only if you cannot use a mutex semaphore. If you must use them, keep critical sections short and avoid including calls that may get blocked.

Handling Floating-Point Exceptions

Floating-point exceptions require special exception handling. In general, you cannot retry a floating-point exception without a significant knowledge of both the 80387 chip and the application that generated the exception. Because knowledge of your application is beyond the capabilities of the VisualAge C++ library, it treats a floating-point exception as a terminating condition.

You can use the `_control87` function and the bit mask values defined in **<float.h>** to mask floating-point exceptions, that is, to prevent them from being reported. Each bit mask corresponds to a unique floating-point exception that can be masked individually. Masking exceptions also changes the state of the floating-point control word for the 80387 chip. When a floating-point exception is masked, the 80387 chip performs a predetermined corrective action.

The bit masks are:

`EM_INVALID`	Mask exceptions resulting from floating-point operations that are not valid. Such an exception can be caused by a floating-point value that is not valid, such as a signalling NaN, or by a problem with the 80387 stack. The corrective action taken by the 80387 chip is to return a quiet NaN. **Note:** Because this type of exception indicates a serious problem, you should not mask it off.
`EM_DENORMAL`	Mask exceptions resulting from the use of denormal floating-point values. The corrective action is to use these values and allow for gradual underflow. This type of exception is not meaningful under the VisualAge C++ compiler and is masked off by default.
`EM_ZERODIVIDE`	Mask the divide-by-zero exception. The 80387 chip returns a value of infinity.
`EM_OVERFLOW`	Mask the overflow exception. The 80387 chip returns a value of infinity.
`EM_UNDERFLOW`	Mask the underflow exception. The 80387 chip returns either a denormal number or zero.
`EM_INEXACT`	Mask the exception that indicates precision has been lost. Because this type of exception is only useful when performing integer arithmetic, while the 80387 chip is used for floating-point arithmetic only, the exception is not meaningful and the 80387 chip ignores it. This exception is masked off by default.

By default, the following bit masks are masked on by default. That is, the exceptions that they correspond to are not masked:

- EM_INVALID
- EM_ZERODIVIDE
- EM_OVERFLOW
- EM_UNDERFLOW

These bit masks are masked off by default. This means that the exceptions that they correspond to are masked by default:

- EM_DENORMAL
- EM_INEXACT

For example, to mask the floating-point underflow exception from being reported, you would code in your source file:

```
oldstate = _control87(EM_UNDERFLOW, EM_UNDERFLOW);
```

To mask it on again, you would code:

```
oldstate = _control87(0, EM_UNDERFLOW);
```

You can also reset the entire floating-point control word to the default state with the `_fpreset` function. Both `_fpreset` and `_control87` are described in the *C Library Reference*.

Important: Because the VisualAge C++ math functions defined in **<math.h>** use the 80387 chip, make sure that when you call any of them, the floating-point control word is set to the default state to ensure exceptions are handled correctly by the VisualAge C++ library.

Note also that the state of the floating-point control word is unique for each thread, and changing it in one thread does not affect any other thread.

Interpreting Machine-State Dumps

Note: This section provides information to be used for Diagnosis, Modification, or Tuning purposes. This information is **not** intended for use as a programming interface.

If you rebuild your program with the `/TI+` option, the IPMD debugger will identify where in the source an exception took place. If the problem does not appear or involves many process or timing problems, you can use the kernel debugger instead. For details on using the kernel debugger, see the *Kernel Debug Reference*.

If you specify the `/Tx+` option, when a process is ended because of an unhandled or incorrectly handled exception, the exception handler performs a machine-state dump. A machine-state dump consists of a number of runtime messages that show information about the state of the system, such as the contents of the registers and the reason for the exception. This information is sent to file handle 2, which is usually associated with stderr. You can also use the _set_crt_msg_handle function to redirect the messages to a file. See the *C Library Reference* for more information about this function.

If you do not specify /Tx+, a message is generated giving the exception and the address at which it occurred.

For example, the following program generates a floating-point exception. Because the exception cannot be handled, a machine-state dump is performed. Figure 24 on page 251 shows what is sent to stderr and explains the messages in the dump.

```
#include <math.h>

int main(void)
{
   _Packed union SIGNAN {       /* a union which allows us to set */
      double dbl;               /*  the parts of a double value   */
      _Packed struct {
         unsigned int siglow : 26;
         unsigned int sighigh : 26;
         unsigned int exp : 11;
         unsigned int sign : 1;
      } dblrep;
   } signan;
   double x;

   /* set the double value to a signalling */
   /* NaN, which the library cannot handle */
   signan.dblrep.sign = 0;
   signan.dblrep.exp = 0x7ff;
   signan.dblrep.sighigh = 0;
   signan.dblrep.siglow = 1;

   /* now call a math function with a */
   /*  signalling NaN to cause a trap */
   x = atan2(signan.dbl,2.0);
```

Figure 23 (Part 1 of 2). Program to Cause a Machine-State Dump

```
   /* the program never gets here */
   return 0;
}
```

Figure 23 (Part 2 of 2). Program to Cause a Machine-State Dump

```
Floating Point Invalid Operation exception occurred at EIP = 00050000 on
 thread 0001.       1
Exception occurred in C Library routine called from EIP = 000112D8.  2
Register Dump at point of exception:  3
EAX = 00000001    EBX = 00000000  ECX = 000B0010    EDX = 00140010
EBP = 00000000    EDI = 00000000  ESI = 00061FCC    ESP = 00061FA8   4
 CS =     005B  CSLIM = 1BFFFFFF   DS =     0053  DSLIM = 1BFFFFFF
 ES =     0053  ESLIM = 1BFFFFFF   FS =     150B  FSLIM = 00000030
 GS =     0000  GSLIM = 00000000   SS =     0053  SSLIM = 1BFFFFFF
NPX Environment:  5
CW = 0362      TW = 3FFF  IP = 005B:0001002B    6
SW = B881  OPCODE = 0545  OP = 0053:00023414
NPX Stack:  7
ST(7): exponent = 0000 significand = + 00000000 00000000    8
Process terminating.    9
```

Figure 24. Example of a Machine-State Dump

1 The first line always states the nature of the exception and the place and thread where the exception occurred. If you specify `/Tx-`, this is the only message that is generated.

2 Indicates that the exception occurred within one of the C library functions. It also indicates the place and thread where the call to that library function was made.

You can use the address given in **1** and **2** to determine where in your code the problem occurred. To do this, you must create a map file by specifying either the compiler option `/B"/map"`, or if you are linking your program separately, the linker option `/map`.

3 Introduces the register dump.

4 Gives the values contained by each register at the time the exception occurred. for information on the purpose of each register, see the documentation for your processor chip.

5 Introduces the state of the numeric processor extension (NPX) at the time of the exception.

6 Gives the values of the elements in the NPX environment.

7 Introduces the state of the NPX stack at the time of the exception.

8 One copy of this message appears for each valid stack entry in the NPX and gives the values for each. In this example, because there is only one stack entry, the message appears only once. If there are no valid stack entries, a different message is issued in place of this message to state that fact.

9 Confirms that the process is terminating. It is one of several informational messages that may accompany the initial exception message and register dump.

In general, a dump will always include items **1**, **3**, and **4**. Item **2** appears only if the exception occurred in a VisualAge C++ library function. Items **5** to **8** appear only if the NPX was in use at the time of the exception. Item **9** may or may not appear, depending on the circumstances of each exception.

For a list of all the runtime messages and their explanations, see the online *Language Reference*.

Note: If you copy and run the program in Figure 23 on page 250, you will get the same messages as shown in Figure 24 on page 251, but the values given may be different.

Common Problems that Generate Exceptions

The following is a list of some of the common problems that can generate runtime exceptions:

- Improper use of memory. For, using a pointer to an object that has already been freed can cause an exception, as can corrupting the heap. In such situations, try rebuilding your program using the Debug Memory option, `/Tm+`.
- Using an invalid pointer.
- Passing an invalid parameter to a system function.
- Return codes from library or system calls that are not checked.

15

Managing Memory

This section describes techniques you can use to manage the memory of your program more efficiently. It includes information on the tiled and debug versions of the memory management functions (like `malloc`), and also tells you how to create and use your own heaps of memory. The runtime functions are described in detail in the *C Library Reference*.

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 C++ has added another function, `_heapmin`, to return unused memory to the system.) VisualAge 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`
Tiled Heap (`/Gt`)	`_tmalloc`	`_debug_tmalloc`

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" on page 258.

Tiled Functions

Use the tiled memory management functions to allocate and free memory from the runtime's tiled memory heap. If you have objects that can be accessed by 16-bit code, you should store them in tiled memory. Tiled memory does not cross 64K boundaries, as long as the object is smaller than 64K. Objects larger than 64K are aligned on 64K boundaries, but will also cross 64K boundaries.

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

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

The names of the tiled versions are prefixed by `_t` (for "tiled"), for example, `_tmalloc`, and they are defined in `<malloc.h>` and `<stdlib.h>`.

The functions provided are:

- `_tcalloc`
- `_tfree`
- `_theapmin`
- `_tmalloc`
- `_trealloc`

You can also create your own heaps of tiled memory. Creating your own heaps is described in "Managing Memory with Multiple Heaps" on page 258.

For more information about sharing objects between 32-bit and 16-bit code, see Chapter 12, "Calling between 32-Bit and 16-Bit Code" on page 191.

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 handle 2 (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 handle 2 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 C Set ++ releases prior to VisualAge 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" on page 273.

Tiled Debug Functions

The tiled functions also have debug versions that work just like the regular and heap-specific debug versions. Use these functions to allocate and free memory from the tiled VisualAge C++ runtime heap. They also provide information that you can use to debug memory problems with the tiled heap.

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 and tiled memory compiler options `/Tm` and `/Gt`, to map calls to the regular memory management functions to their tiled 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 tiled debug versions are prefixed by `_debug_t`, for example, `_debug_tmalloc`, and they are defined in `<malloc.h>` and `<stdlib.h>`.

The functions provided are:

- `_debug_tcalloc`
- `_debug_tfree`
- `_debug_theapmin`
- `_debug_tmalloc`
- `_debug_trealloc`
- `_tdump_allocated`
- `_tdump_allocated_delta`
- `_theap_check`

For more information about debugging memory problems, see "Debugging Your Heaps" on page 273.

Managing Memory with Multiple Heaps

VisualAge C++ now gives you the option of creating and using your own pools of memory, called *heaps*. You can use your own heaps in place of or in addition to the default VisualAge C++ runtime heap to improve the performance of your program. This section describes how to implement multiple user-created heaps using VisualAge C++.

Note: Many readers will not be interested in creating their own heaps. Using your own heaps is entirely optional, and your applications will work perfectly well using the default memory management provided (and used by) the VisualAge C++ runtime library. If you want to improve the performance and memory management of your program, multiple heaps can help you. Otherwise, you can ignore this section and any heap-specific library functions.

Why Use Multiple Heaps?

Using a single runtime heap is fine for most programs. However, using multiple heaps can be more efficient and can help you improve your program's performance and reduce wasted memory for a number of reasons:

- When you allocate from a single heap, you may end up with memory blocks on different pages of memory. For example, you might have a linked list that allocates memory each time you add a node to the list. If you allocate memory for other data in between adding nodes, the memory blocks for the nodes could end up on many different pages. To access the data in the list, the system may have to swap many pages, which can significantly slow your program.

  With multiple heaps, you can specify which heap you allocate from. For example, you might create a heap specifically for the linked list. The list's memory blocks and the data they contain would remain close together on fewer pages, reducing the amount of swapping required.

- In multithread applications, only one thread can access the heap at a time to ensure memory is safely allocated and freed. For example, say thread 1 is allocating memory, and thread 2 has a call to `free`. Thread 2 must wait until thread 1 has finished its allocation before it can access the heap. Again, this can slow down performance, especially if your program does a lot of memory operations.

  If you create a separate heap for each thread, you can allocate from them concurrently, eliminating both the waiting period and the overhead required to serialize access to the heap.

- With a single heap, you must explicitly free each block that you allocate. If you have a linked list that allocates memory for each node, you have to traverse the entire list and free each block individually, which can take some time.

  If you create a separate heap for that linked list, you can destroy it with a single call and free all the memory at once.

- When you have only one heap, all components share it (including the VisualAge C++ runtime library, vendor libraries, and your own code). If one component corrupts the heap, another component might fail. You may have trouble discovering the cause of the problem and where the heap was damaged.

  With multiple heaps, you can create a separate heap for each component, so if one damages the heap (for example, by using a freed pointer), the others can continue unaffected. You also know where to look to correct the problem.

You can create heaps of regular memory, tiled memory, or shared memory, and you can have any number of heaps of any type. (See “Types of Memory” on page 265 for more information about the different types of memory for heaps.) The only limit is the space available on your operating system (your machine's memory and swapper size, minus the memory required by other running applications).

VisualAge C++ provides heap-specific versions of the memory management functions (`malloc` and so on), and a number of new functions that you can use to create and manage your own heaps of memory. Debug versions of all the memory management functions are available, including the heap-specific ones. You can also use debug versions with tiled memory and shared memory, which you couldn't do in previous releases.

Note: Because multiple heaps and the functions that support them are extensions to the ANSI language standard, you can only use them when the language level is set to extended (with the `/Se` compiler option or `#pragma langlvl(extended)` directive).

The following sections describe how to create and use your own heaps. For detailed information on each function, refer to the *C Library Reference*.

Creating a Fixed-Size Heap

Before you create a heap, you need to get the block of memory that will make up the heap. You can get this block by calling an OS/2 API (such as `DosAllocMem` or `DosAllocSharedMem`) or by statically allocating it.

Make sure the block is large enough to satisfy all the memory requests your program will make of it, as well as the internal information for managing the heap. Once the block is fully allocated, further allocation requests to the heap will fail.

The internal information requires _HEAP_MIN_SIZE bytes (_HEAP_MIN_SIZE is defined in **<umalloc.h>**); you cannot create a heap smaller than this. Add the amount of memory your program requires to this value to determine the size of the block you need to get.

Also make sure the block is the correct type (regular, tiled, or shared) for the heap you are creating.

Once you have the block of memory, create the heap with `_ucreate`.

For example:

```
Heap_t fixedHeap;    /* this is the "heap handle" */
/* get memory for internal info plus 5000 bytes for the heap */
static char *block[_HEAP_MIN_SIZE + 5000];

fixedHeap = _ucreate(block, (_HEAP_MIN_SIZE+5000),  /* block to use */
                     !_BLOCK_CLEAN,   /* memory is not set to 0   */
                     _HEAP_REGULAR,   /* regular memory           */
                     NULL, NULL);     /* we'll explain this later */
```

The !_BLOCK_CLEAN parameter indicates that the memory in the block has not been initialized to `0`. If it were set to `0` (for example, by `DosAllocMem` or `memset`), you would specify _BLOCK_CLEAN. The `calloc` and `_ucalloc` functions use this information to improve their efficiency; if the memory is already initialized to `0`, they don't need to initialize it.

Note: `DosAllocMem` initializes memory to `0` for you. You can also use `memset` to initialize the memory; however, `memset` also commits all the memory at once, which could slow overall performance.

The fourth parameter indicates what type of memory the heap contains: regular (_HEAP_REGULAR), tiled (_HEAP_TILED), or shared (_HEAP_SHARED). The different memory types are described in “Types of Memory” on page 265.

For a fixed-size heap, the last two parameters are always NULL.

Using Your Heap

Once you have created your heap, you need to open it for use by calling `_uopen`:

```
_uopen(fixedHeap);
```

This opens the heap for that particular process; if the heap is shared, each process that uses the heap needs its own call to `_uopen`.

You can then allocate and free from your own heap just as you would from the default heap. To allocate memory, use `_ucalloc` or `_umalloc`. These functions work just like `calloc` and `malloc`, except you specify the heap to use as well as the size of block that you want. For example, to allocate 1000 bytes from `fixedHeap`:

```
void *up;
up = _umalloc(fixedHeap, 1000);
```

To reallocate and free memory, use the regular `realloc` and `free` functions. Both of these functions always check what heap the memory came from, so you don't need to specify the heap to use. For example, in the following code fragment:

```
void *p, *up;
p = malloc(1000);   /* allocate 1000 bytes from default heap */
up = _umalloc(fixedHeap, 1000);  /* allocate 1000 from fixedHeap */

realloc(p, 2000);   /* reallocate from default heap */
realloc(up, 100);   /* reallocate from fixedHeap    */

free(p);            /* free memory back to default heap */
free(up);           /* free memory back to fixedHeap    */
```

the `realloc` and `free` calls look exactly the same for both the default heap and your heap.

For any object, you can find out what heap it was allocated from by calling `_mheap`. You can also get information about the heap itself by calling `_ustats`, which tells you:

- How much memory the heap holds (excluding memory used for overhead)
- How much memory is currently allocated from the heap
- What type of memory is in the heap
- The size of the largest contiguous piece of memory available from the heap

When you call any heap function, make sure the heap you specify is valid. If the heap is not valid, the behavior of the heap functions is undefined.

Adding to a Fixed-Size Heap

Although you created the heap with a fixed size, you can add blocks of memory to it with `_uaddmem`. This can be useful if you have a large amount of memory that is allocated conditionally. Like the starting block, you must first get the block to add to the heap by using an OS/2 API or by allocating it statically. Make sure the block you add is the same type of memory as the heap you are adding it to.

For example, to add 64K to `fixedHeap`:

```
void *newblock;
/* get memory block from operating system */
DosAllocMem(newblock, 65536, PAG_COMMIT | PAG_WRITE | PAG_READ);

_uaddmem(fixedHeap,       /* heap to add to */
         newblock, 65536, /* block to add   */
         _BLOCK_CLEAN);   /* DosAllocMem sets memory to 0 */
```

Using `_uaddmem` is the only way to increase the size of a fixed heap.

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.

Destroying Your Heap

When you have finished using the heap, close it with `_uclose`. Once you have closed the heap in a process, that process can no longer allocate from or return memory to that heap. If other processes share the heap, they can still use it until you close it in each of them. Performing operations on a heap after you've closed it causes undefined behavior.

To finally destroy the heap, call `_udestroy`. If blocks of memory are still allocated somewhere, you can force the destruction. Destroying a heap removes it entirely even if it was shared by other processes. Again, performing operations on a heap after you've destroyed it causes undefined behavior.

After you destroy your fixed-size heap, it is up to you to return the memory for the heap (the initial block of memory you supplied to `_ucreate` and any other blocks added by `_uaddmem`) to the system.

Creating an Expandable Heap

With a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. In this section, we will create a heap that can expand and contract.

With the VisualAge C++ runtime heap, when not enough storage is available for your `malloc` request, the runtime gets additional storage from the system. Similarly, when you minimize the heap with `_heapmin` or when your program ends, the runtime returns the memory to the operating system.

When you create an expandable heap, you provide your own functions to do this work (we'll call them *`getmore_fn`* and *`release_fn`*, although you can name them whatever you choose). You specify pointers to these functions as the last two parameters to `_ucreate` (instead of the NULL pointers you used to create a fixed-size heap). For example:

```
Heap_t growHeap;
static char *block[_HEAP_MIN_SIZE];  /* get block */

growHeap = _ucreate(block, _HEAP_MIN_SIZE,   /* starting block */
                    !_BLOCK_CLEAN,      /* memory not set to 0 */
                    _HEAP_REGULAR,      /* regular memory      */
                    getmore_fn,     /* function to expand heap */
                    release_fn);    /* function to shrink heap */
```

Note: You can use the same *getmore_fn* and *release_fn* for more than one heap, as long as the heaps use the same type of memory and your functions are not written specifically for one heap.

Expanding Your Heap

When you call `_umalloc` (or a similar function) for your heap, `_umalloc` tries to allocate the memory from the initial block you provided to `_ucreate`. If not enough memory is there, it then calls your *getmore_fn*. Your *getmore_fn* then gets more memory from the operating system and adds it to the heap. It is up to you how you do this.

Your *getmore_fn* must have the following prototype:

```
void *(*getmore_fn)(Heap_t uh, size_t *size, int *clean);
```

The *uh* is the heap to be expanded.

The *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*, reducing your program's execution speed. We recommend you return multiples of 64K (the smallest size that `DosAllocMem` returns). Make sure that you update the *size* parameter. if you return more than the *size* requested.

Your function must also set the *clean* parameter to either _BLOCK_CLEAN, to indicate the memory has been set to `0`, or !_BLOCK_CLEAN, to indicate that the memory has not been initialized.

The following fragment shows an example of a *getmore_fn*:

```
static void *getmore_fn(Heap_t uh, size_t *length, int *clean)
{
   char *newblock;

   /* round the size up to a multiple of 64K */
   *length = (*length / 65536) * 65536 + 65536;

   DosAllocMem(&newblock, *length, PAG_COMMIT | PAG_READ | _PAG_WRITE);

   *clean = _BLOCK_CLEAN;  /* mark the block as "clean" */
   return(newblock);       /* return new memory block   */
}
```

Note: Be sure that your *getmore_fn* allocates the right type of memory (regular, tiled, or shared) for the heap. There are also special considerations for shared memory, described under "Types of Memory" on page 265.

You can also use _uaddmem to add blocks to your heap, as you did for the fixed heap in "Adding to a Fixed-Size Heap" on page 261. _uaddmem works exactly the same way for expandable heaps.

Shrinking Your Heap

To coalesce the heap (return all blocks in the heap that are totally free to the system), use _uheapmin. _uheapmin works like _heapmin, except that you specify the heap to use.

When you call _uheapmin to coalesce the heap or _udestroy to destroy it, these functions call your *release_fn* to return the memory to the system. Again, it is up to you how you implement this function.

Your *release_fn* must have the following prototype:

```
void (*release_fn)(Heap_t uh, void *block, size_t size);
```

Where *uh* identifies the heap to be shrunk. The pointer *block* and its *size* are passed to your function by _uheapmin or _udestroy. Your function must return the memory pointed to by *block* to the system. For example:

```
static void release_fn(Heap_t uh, void *block, size_t size)
{
   DosFreeMem(block);
   return 0;
}
```

Notes:

1. `_udestroy` calls your *`release_fn`* to return all memory added to the *`uh`* heap by your *`getmore_fn`* or by `_uaddmem`. However, you are responsible for returning the initial block of memory that you supplied to `_ucreate`.
2. Because a fixed-size heap has no *`release_fn`*, `_uheapmin` and `_udestroy` work slightly differently. Calling `_uheapmin` for a fixed-size heap has no effect but does not cause an error; `_uheapmin` simply returns `0`. Calling `_udestroy` for a fixed-size heap marks the heap as destroyed, so no further operations can be performed on it, but returns no memory. It is up to you to return the heap's memory to the system.

Types of Memory

There are three different types of memory:

1. Regular

   Most programs use regular memory. This is the type provided by the default runtime heap.

2. Tiled

   Tiled memory is guaranteed not to cross 64K boundaries (as long as the object being allocated is less than 64K), so it is appropriate for objects that may be accessed by 16-bit code. If you want to use tiled memory, make sure you specify that type when you get the initial block for your heap, when you create the heap, and when you add to your heap (with `_uaddmem` or your *`getmore_fn`*). Even if you use tiled memory, you still need to specify the `/Gt` compiler option to ensure your objects are correctly aligned.

   Note: When you use /Gt+, the regular versions of the memory management functions (that use the default heap) are mapped to tiled versions so that they also return tiled memory. If you parenthesize the function calls, they are **not** mapped to the tiled versions and therefore return regular memory. However, if you replace the default runtime heap with your own tiled heap, all regular memory management functions will use the new default heap of tiled memory, regardless of whether the calls are parenthesized. There are no tiled versions of the heap-specific functions; to use tiled memory, create a tiled heap and specify it when you call the functions.

3. Shared

   Heaps of shared memory can be shared between processes or applications. If you want other processes to use the heap you have created, you must pass them the heap handle and give them access to the heap.

   To correctly share a heap, you must observe certain requirements, described below. Meeting these requirements is entirely your responsibility.

For a fixed-size shared heap, you must:

- Allocate the starting block of memory as shared memory.
- Ensure that all processes that use the heap (allocate or free memory from it) or that use shared data from the heap ("interested" processes) have access to the starting block.
- Pass the heap handle to each process that uses the heap ("interested" processes do not need the heap handle).
- Call `_uopen` for the heap in each process that uses it.
- Call `_uclose` in each process where you called `_uopen` before you destroy the heap.

If you use `_uaddmem` to expand your heap, you must ALSO:

- Ensure that all interested and using processes have access to the new block added by `_uaddmem`.

If you use *getmore_fn* and *release_fn* to dynamically expand and shrink your heap, you must ALSO:

- Ensure that all interested and using processes have access to the new block added by *getmore_fn*, no matter which process called *getmore_fn*.
- Ensure that *getmore_fn* can run successfully in any process that may call a heap-specific allocation function (such as `_umalloc`) or cause it to be called; and that *release_fn* can run successfully in any process that may call `_uheapmin` or `_udestroy` or cause them to be called. Typically this would mean placing both *getmore_fn* and *release_fn* in a DLL that is loaded by all processes that use the heap handle. Any data these functions use must also be shared.
- Ensure that *release_fn* revokes the access of each process to a shared memory block before it returns that block to the system.

For more information about managing shared memory and OS/2 APIs you can use, see the section on Shared Memory in the *Control Program Guide and Reference*.

Changing the Default Heap

The regular memory management functions (`malloc` and so on) always use whatever heap is currently the default for that thread. The initial default heap for all VisualAge C++ applications is the runtime heap provided by VisualAge C++. However, you can make your own heap the default by calling `_udefault`. Then all calls to the regular memory management functions allocate from your heap instead of the runtime heap.

The default heap changes only for the thread where you call `_udefault`. You can use a different default heap for each thread of your program if you choose.

This is useful when you want a component (such as a vendor library) to use a heap other than the VisualAge C++ runtime heap, but you can't actually alter the source code to use heap-specific calls. For example, if you set the default heap to a tiled or shared heap then call a library function that calls `malloc`, the library allocates storage in tiled or shared memory.

Because `_udefault` returns the current default heap, you can save the return value and later use it to restore the default heap you replaced. You can also change the default back to the VisualAge C++ runtime heap by calling `_udefault` and specifying _RUNTIME_HEAP (defined in **<malloc.h>**). You can also use this macro with any of the heap-specific functions to explicitly allocate from the runtime heap.

A Simple Example of a User Heap

The following program shows very simply how you might create and use a heap.

```
/* the function _umalloc calls to get more storage */
static void *get_fn(Heap_t uh, size_t *length, int *clean)
{
   char *p;

   /* DosAllocMem sets storage to 0, so it is "clean" */
   *clean = _BLOCK_CLEAN;

   /* round the block size to a multiple of 64K for efficiency */
   *length = (*length / 65536) * 65536 + 65536;

   /* get the storage from the system)
   DosAllocMem(&p, *length, PAG_COMMIT|PAGE_READ|PAGE_WRITE);

   return p;
}
```

Figure 25 (Part 1 of 3). Example of a User Heap

```
/* the function _heapmin and _destroy
   call to return storage to the system      */
static void release_fn(Heap_t uh, void *p, size_t size)
{
   DosFreeMem(p);
   return;
}

int main(void)
{
   Heap_t myheap;
   /* startchunk will be the first block of storage on the heap */
   char startchunk[_HEAP_MIN_SIZE];
   void *p;

   /* create a heap starting with the block declared earlier */
   myheap = _ucreate(startchunk, _HEAP_MIN_SIZE,
                     !_BLOCK_CLEAN,    /* memory is not set to 0 */
                     _HEAP_REGULAR,    /* regular memory         */
                     get_fn, release_fn);

   if (myheap == NULL)              /* check that valid heap was created */
      puts("create failed");

   if (_open(myheap))            /* open heap and check for failure   */
      puts("open failed");

   /* allocate from myheap; if necessary, _umalloc calls get_fn */
      _umalloc calls get_fn
   p = _umalloc(myheap, 1000);
   if (p == NULL)            /* check that allocation worked      */
      puts("allocation failed");

   p = realloc(p, 100);    /* reallocate from myheap - realloc knows
                              what heap the storage came from      */
   free(p);   /* free also knows what heap the storage came from   */

   /* return unused blocks to system; _heapmin calls release_fn */
   _uheapmin(myheap);
```

Figure 25 (Part 2 of 3). Example of a User Heap

```
   _uclose(myheap);     /* close myheap */

   /* destroy myheap with FORCE because some storage is still
      allocated; _udestroy calls release_fn to return storage */
   _udestroy(myheap, FORCE);
   return 0;
}
```

Figure 25 (Part 3 of 3). Example of a User Heap

A More Complex Example Featuring Shared Memory

The following program shows how you might implement a heap shared between a parent and several child processes.

Figure 26 shows the parent process, which creates the shared heap. First the main program calls the `init` function to allocate shared memory from the operating system (using `DosAllocSharedMem`) and name the memory so that other processes can use it by name. The `init` function then creates and opens the heap. The loop in the main program performs operations on the heap, and also starts other processes. The program then calls the `term` function to close and destroy the heap.

```
#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define INCL_DOSMEMMGR
#include <os2.h>
```

Figure 26 (Part 1 of 4). Example of a Shared User Heap - Parent Process

```
/* Function to create and open the heap with a named shared memory object */
static int inithp(Heap_t *uheap, void **init_block)
{
   int flags;                           /* to store flags for DosAllocMem */
   void *p;                             /* to point to the shared memory  */
   const size_t size = 65536;                   /* always allocate 64K */
   flags = PAG_COMMIT | PAG_WRITE | PAG_READ;

   /* allocate shared memory from system */
   if (DosAllocSharedMem(&p,              /* assign memory block to p */
                         "\\SHAREMEM\\MYNAME",        /* name memory block */
                         size, flags))
      return 1;
   else
   {
      *uheap = _ucreate((char *)p + sizeof(Heap_t), /* heap handle goes at start */
                     size - sizeof(Heap_t), /*block size,space for heap handle */
                     _BLOCK_CLEAN,      /* DosAllocSharedMem sets memory to 0 */
                     _HEAP_SHARED | _HEAP_REGULAR, /* shared or regular heap */
                     NULL, NULL);                         /* fixed size */

      if (*uheap = NULL)                          /* check heap was created */
         return 1;
      memcpy(p, uheap, sizeof(Heap_t)); /* store heap handle in shared area */
   }

   if (_uopen(*uheap))                      /*open heap and check result */
      return 1;
   *init_block = p;          /*make initial block point to shared memory */
   return 0;
}

/* Function to close and destroy the heap */
static int term(Heap_t uheap, void init_block)
{
   if (_uclose(uheap))                                     /* close heap */
      return 1;
```

Figure 26 (Part 2 of 4). Example of a Shared User Heap - Parent Process

```
   if (_udestroy(uheap, FORCE))              /* force destruction of heap */
      return 1;

   DosFreeMem(init_block);                     /* return memory to system */
   return 0;
}

int main(void)
{
   int rc, a;                            /* for return codes, loop iteration */
   Heap_t uheap;                              /* heap to create          */
   void *init_block;                          /* initial block to use     */
   char *p;                                      /* for allocating from heap */

   /* call init function to create and open the heap  */
   rc = init(&uheap, &init_block);

   if (rc)                             /* check for success (0) or failure (1) */
      return rc;                            /*  if failure, program ends       */

   /* perform operations on uheap */
   for (a = 1; a < 10; a++)
   {
      p = _umalloc(uheap, 10);                        /* allocate from uheap */
      if (p == NULL)
         return 1;
      memset(p, 'M', _msize(p));             /* set all bytes in p to 'M' */

      if (system("sample1b.exe"))    /*start new process and check result */

      p = realloc(p,50);                           /* reallocate from uheap */
      if (p == NULL)
         return 1;
      memset(p, 'R', _msize(p));          /* set all bytes in p to 'R' */
   }
```

Figure 26 (Part 3 of 4). Example of a Shared User Heap - Parent Process

```
   /* call term function to close and destroy the heap */
   rc = term(uheap, init_block);

   puts("Sample 1 ending...");
   return rc;
}
```

Figure 26 (Part 4 of 4). Example of a Shared User Heap - Parent Process

Figure 27 shows the process started by the loop in the parent process. This process uses `DosGetSharedMem` to access the shared memory by name, then extracts the heap handle for the heap created by the parent process. The process then opens the heap, makes it the default heap, and performs some operations on it in the loop. After the loop, the process replaces the old default heap, closes the user heap, and ends.

```
#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define INCL_DOSMEMMGR
#include <os2.h>

int main(void)
{
   int rc, a;        /* for return code, loop iteration */
   Heap_t uheap, oldheap;         /* heap to create, old default heap */
   void *init_block;               /* to point to the shared memory   */
   char *p;                           /* for allocating from the heap    */

   /* get the named shared memory block and check result */
   if (DosGetNamedSharedMem(&init_block, /* assign to init_block */
                           "\\SHAREMEM\\MYNAME",        /* name of memory */
                           PAG_READ | PAG_WRITE))     /* flags */
      return 1;
```

Figure 27 (Part 1 of 2). Example of a Shared User Heap - Child Process

```
   /* extract heap handle from shared memory and assign to uheap */
   memcpy(&uheap, init_block, sizeof(Heap_t));

   if (_uopen(uheap))                      /* open heap and check result */
      return 1;

   /* register uheap as default runtime heap, save old default */
   oldheap = _udefault(uheap);
   if (oldheap == NULL)
      return 1;

   /* perform operations on uheap */
   for (a = 1; a < 10; a++)
   {
      p = malloc(10);      /* malloc uses default heap, which is now uheap */
      memset(p, 'M', _msize(p));
      free(p);
   }

   /* replace original default heap and check result */
   if (uheap != _default(oldheap))
      return 1;

   if (_uclose(uheap))             /* close the heap for this process */
      return 1;

   return 0;
```

Figure 27 (Part 2 of 2). Example of a Shared User Heap - Child Process

Debugging Your Heaps

VisualAge C++ provides two sets of functions for debugging your memory problems:

1. Debug versions of all memory management functions
2. Heap-checking functions similar to those provided by other compilers.

Debug Memory Management Functions

Debug versions of the heap-specific memory management functions are provided, just as they are for the regular versions. Each debug version performs the same function as its non-debug counterpart. In addition, the debug version calls `_uheap_check` to check the heap used in the call, and records the file and line number where the memory was allocated or freed. You can then use `_dump_allocated` or `_dump_allocated_delta` to display information about currently allocated memory blocks. Information is printed to **stderr**; you can change the destination with the `_set_crt_msg_handle` function.

You can use debug memory management functions for any type of heap, including tiled and shared memory. To use the debug versions, specify the Debug Memory compiler option, `/Tm`. The VisualAge C++ compiler then maps all calls to memory management functions (regular or heap-specific) to the corresponding debug version.

Note: If you parenthesize the name of a memory management function, the function is **not** mapped to the debug version. This does not apply to the C++ `new` and `delete` functions, which are mapped to their debug versions regardless of parentheses.

Prior to Version 2.1 of VisualAge C++, debug memory management functions and tiled memory functions (for example, `_tmalloc`) were not compatible. You can now specify both `/Gt` and `/Tm` together, to map regular memory management functions to tiled debug versions (for example, `_debug_tmalloc`).

Another restriction in earlier versions of VisualAge C++ was that you could not mix debug and non-debug memory management functions, meaning you could not allocate with a regular function and then free the object with the debug version. This restriction no longer exists.

Skipping Heap Checks

As stated above, each debug function calls `_heap_check` (or `_uheap_check`) to check the heap. Although this is useful, it can also increase your program's memory requirements and decrease its execution speed.

To reduce the overhead of checking the heap on every debug memory management function, you can control how often the functions check the heap with the DDE4_HEAP_SKIP environment variable. This is not required in most applications unless the application is extremely memory intensive.

Set DDE4_HEAP_SKIP with the `SET` command, like any other environment variable. You can set it either in CONFIG.SYS, from the command line, or in your project. The syntax for DDE4_HEAP_SKIP is:

```
SET DDE4_HEAP_SKIP= increment, [start]
```

increment specifies how often you want the debug functions to check the heap. In the above statement, the comma is optional. The *start* parameter is also optional; you can use it to start skipping heap checks after *start* calls to debug functions.

When you use *start* parameter to start skipping heap checks, you are trading off heap checks that are done implicitly against program execution speed. You should therefore start with a small increment (like 5) and slowly increase until the application is usable.

For example, if you specify:

```
SET DDE4_HEAP_SKIP= 10
```

then every tenth debug memory function call performs a heap check. If you specify:

```
SET DDE4_HEAP_SKIP= 5, 100
```

then after 100 debug memory function calls, only every fifth call performs a heap check. Other than the heap check, the debug functions behave exactly the same as usual.

Heap-Checking Functions

VisualAge C++ also provides some new functions for validating user heaps: `_uheapchk`, `_uheapset`, and `_uheap_walk` (Each of these functions has a non-heap-specific version that validates the default heap.)

Both `_uheapchk` and `_uheapset` check the specified heap for minimal consistency; `_uheapchk` checks the entire heap, while `_uheapset` checks only the free memory. `_uheapset` also sets the free memory in the heap to a value you specify. `_uheap_walk` traverses the heap and provides information about each allocated or freed object to a callback function that you provide. You can then use the information however you like.

These heap-checking functions are defined in **<umalloc.h>** (the regular versions are also in **<malloc.h>**). They are not controlled by a compiler option, so you can use them in your program at any time.

Which Should I Use?

Both sets of debugging functions have their benefits and drawbacks. Which you choose to use depends on your program, your problems, and your preference.

The debug memory management functions provide detailed information about all allocation requests you make with them in your program. You don't need to change any code to use the debug versions; you need only specify the `/Tm` compiler option. However, because only calls that have been mapped to debug versions provide any information, you may have to rebuild many or all of your program's modules, which can be time-consuming.

On the other hand, the heap-checking functions perform more general checks on the heap at specific points in your program. You have greater control over where the checks the occur. The heap-checking functions also provide compatibility with other compilers that offer these functions. You only have to rebuild the modules that contain the heap-checking calls. However, you have to change your source code to include these calls, which you will probably want to remove in your final code. Also, the heap-checking functions only tell you if the heap is consistent or not; they do not provide the details that the debug memory management functions do.

What you may choose to do is add calls to heap-checking functions in places you suspect possible memory problems. If the heap turns out to be corrupted, at that point you may want to rebuild with the `/Tm` option.

Note: When the debug memory option `/Tm` is specified, code is generated to *pre-initialize* the local variables for all functions. This makes it much more likely that uninitialized local variables will be found during the normal debug cycle rather than much later (usually when the code is optimized).

Regardless of which debugging functions you choose, your program requires additional memory to maintain internal information for these functions. If you are using fixed-size heaps, you may have to increase the heap size in order to use the debugging functions.

For more information on the debug memory management functions and how to use them, see “Debug Functions” on page 255.

16 The IBM System Object Model

The IBM System Object Model (SOM) provides a common programming interface for building and using objects. SOM improves your C++ programming productivity in two ways:

- If you develop or maintain libraries of C++ classes and methods that are used by other application developers, SOM allows you to release new versions of a library without requiring users of the library to recompile their applications.
- SOM lets you make your C++ classes and objects accessible to programs written in other languages, and to write C++ programs that use classes and objects created using other SOM-supported languages.

You can make classes and methods in existing C++ programs SOM-accessible without having to rewrite class and method definitions. Although SOM imposes some restrictions on C++ coding conventions, you should be able to convert most C++ programs for SOM support with minimal effort. VisualAge C++ can convert existing C++ classes to SOM classes. This method of creating SOM classes is sometimes referred to as the Direct-to-SOM or DTS method, and a Direct-to-SOM or DTS class is one that has been converted to SOM by the compiler.

For information on how you can have the compiler convert classes to SOM, see "Converting C++ Programs to SOM Using SOMAsDefault" on page 307 and "Creating SOM-Compliant Programs by Inheriting from SOMObject" on page 308.

This chapter does not describe the entire scope of SOM. For more detail on SOM, see the online *SOM Programming Guide* and the online *SOM Programming Reference*.

What is SOM?

SOM defines an interface between programs, or between libraries and programs, so that an object's interface is separated from its implementation. SOM allows classes of objects to be defined in one programming language and used in another, and it allows libraries of such classes to be updated without requiring client code to be recompiled.

A SOM library consists of a set of classes, methods, static functions, and data members. Programs that use a SOM library can create objects of the types defined in the library, use the methods defined for an object type, and derive subclasses from SOM classes, even if the language of the program accessing the SOM library does not support class typing. A SOM library and the programs that use objects and methods of that library need not be written in the same programming language. SOM also minimizes the impact of revisions to libraries. If a SOM library is changed to add new classes or methods, or to change the internal implementation of classes or methods, you can still run a program that uses that library without recompiling. This is not the case for all other C++ libraries, which in some cases require recompilation of all programs that use them whenever the libraries themselves are changed.

SOM provides an Application Programming Interface (API) that gives programs access to information about a SOM class or SOM object. Any SOM class inherits a set of virtual methods that can be used, for example, to find the class name of an object, or to determine whether a particular method is available for an object. These API functions are fully described in the online *SOM Programming Guide*.

You can make your C++ classes and methods SOM-accessible in one of two ways: by using pragmas to direct the compiler in generating a SOM interface for your code, or by explicitly deriving your classes from SOMObject. Both of these techniques are described later in this chapter. In both cases, VisualAge C++ can also generate Interface Definition Language (IDL) files which are required to make your C++ SOM classes accessible to non-C++ programs. For further details see "Interface Definition Language (IDL) Considerations" on page 291.

Once you have a SOM-compliant version of your library, you can add methods, types, and subtypes to that library, or change the implementation of methods, without requiring programs that use your library to be recompiled. These programs need only be recompiled if they themselves are modified, for example to make use of newly defined types or methods. See "SOM and Upward Binary Compatibility of Libraries" on page 280 for further details.

SOM and CORBA

SOM complies with the Common Object Request Broker Architecture (CORBA) standard defined by the Object Management Group. CORBA is an industry-wide standard for the management of objects across heterogeneous, distributed systems.

The Cost of Using SOM

SOM is a powerful tool, but the flexibility that it gives you comes at a price. A program that is SOM-enabled may run more slowly than an equivalent one in native C++. You should weigh the many benefits of SOM against the negative effect it may have on the performance of your program.

SOM and DSOM

Distributed SOM (DSOM) is an extension of SOM that permits the creation of client programs capable of calling the methods of remote SOM objects. Such method calls are entirely transparent to both the client and the server. When you compile your C++ classes with SOM support, those classes can be used/in DSOM applications.

For further details on DSOM, see “Using SOM Classes in DSOM Applications” on page 309.

What is DTS?

If you are an experienced SOM programmer who has used earlier versions of SOM from C or C++ programs, you know that SOM defines *bindings* for those languages. The language bindings consist of a number of macros plus structure or class definitions in header files with the extensions **.h** and **.ih** (for C) and **.xh** and **xih** (for C++). They are generated for a particular SOM class by running the SOM Compiler **sc.exe** on the **.idl** file for that class interface. The bindings can be used with a wide range of C and C++ compilers and do not require special compiler support.

Direct-to-SOM (DTS) is a new and much more flexible way of using SOM in a C++ program. DTS class definitions resemble regular C++ classes, and you can either write them directly or use **sc.exe** to generate them into files with an **.hh** extension from existing IDL. DTS C++ class definitions can only be used with C++ compilers like VisualAge C++ that support DTS.

DTS provides the same access to SOM functionality that the C++ bindings do but, in addition, DTS supports far more of the C++ language. DTS supports member operators, conversion functions, user-defined **new** and **delete** operators, function overloading, stack local SOM objects, and first-class source debugging support for SOM classes. You can write and subclass your DTS classes directly and may never need to write a line of IDL.

VisualAge C++ supports DTS C++, but still can be used with C and C++ bindings. SOM DLLs and EXEs can interoperate freely whether constructed using C bindings, C++ bindings, or DTS C++.

Warning: Within one single C++ compilation, it is **not** possible to use both C++ bindings *and* DTS. A useful rule of thumb is that if you include any **.xh** header files in your compilation, you must not also include any **.hh** files, or use the **SOMAsDefault** pragma or the /Ga option.

Interface Definition Language

The Interface Definition Language (IDL) is a language-independent notation for specifying the interfaces of SOM objects. It is required for implementing DSOM classes, and when making your C++ SOM classes accessible from other languages. VisualAge C++ generates an IDL description of your SOM classes for you. For more information about IDL, see the online *SOM Programming Guide*.

SOM and Upward Binary Compatibility of Libraries

This section is intended for programmers who are developing or maintaining libraries containing C++ class and object definitions. This section does not describe how to write programs that *use* a SOM-compliant library.

When you make changes to a SOM library that contains C++ class and method definitions, programs that use your library may or may not need to be recompiled in order to work with the new version of the library. Changes to your library that *may not* require recompilation of client programs include:

- Adding new classes, including base classes
- Adding new methods or data members to existing classes
- Changing or removing private methods or data members from classes
- Changing the internal implementation of public or protected methods
- Moving member functions from a derived class to a base class.

For more detail on such changes, see the online *SOM Programming Guide* and the online *SOM Programming Reference*.

If you change your library only in the ways described above, and you follow the rules described in Release Order of SOM Objects, you can provide the new library to your users in binary form, and their programs will work with the new library without needing to be recompiled, or even relinked if the library is a dynamically linked library.

Changes to your library that *will* require recompilation of client programs include:

- Removing classes
- Removing public data members, methods, or static member functions from existing classes.

In the context of the above list, removing also includes renaming. Renaming an item from a library is equivalent to removing the item and adding a new item with the same characteristics. If you use the **SOMMethodName** or **SOMClassName** pragmas to provide a SOM name for a C++ method or class, changing the SOM name has the same effect as renaming the C++ method or class name.

Note that if you add the **SOMMethodName** or **SOMNoMangling** pragmas for a method this also changes the SOM name from that supplied by the compiler to that specified by the pragma. If there is any likelihood of non-C++ programs using your SOM classes, use these pragmas for your initial implementation.

The remainder of this section describes details of how SOM provides upward binary compatibility of libraries. You do not need to know this information to create or maintain SOM-compliant libraries, but the information will help you understand when and why certain SOM pragmas are used (specifically, **SOMReleaseOrder** and **SOMClassVersion**).

Release Order of SOM Objects

SOM achieves binary compatibility by arranging all the components of a class into ordered lists, locating them by their position in a list, and by enforcing rules to ensure that the ordering of the lists never changes. There are three lists maintained for each class. Two lists are for instance data, and one is for member functions.

The first list is for public instance data. The ordering in this list is the declaration order of the public instance data in the class. The corresponding rule that preserves this order and ensures binary upward compatibility is that the declaration order must not change, and that new public data members must be added after all preexisting public members.

The second list is for protected and private instance data. This list is ordered and the order preserved in exactly the same manner as for the public instance data list.

Adding new public or protected data members only forces you to recompile clients that need to use the new data.

Deleting or reordering public data members will break binary compatibility, and require recompilation of all clients and derived classes. Deleting or reordering protected data members will require recompilation of derived classes, but not of clients since they did not have access to the protected data.

The third ordered list is a list of all member functions introduced by the class (both static and nonstatic), plus any static data members in the class.

Virtual functions that override virtual functions in base classes do not appear in this list, but do appear in the list belonging to the base class that introduced them. As a special case of this rule, a class's default constructor, copy constructor, destructor, and default assignment operator are all treated as overrides of virtual functions introduced by SOMObject, and so do not appear in the derived class's list.

This third list, called the "release order", is determined in one of two ways. The simpler way is the declaration order of the member functions and static data members, and the resulting compatibility rule is that once again new members must be added after all others in the class declaration. Note that for the purposes of this rule, attributes created using the **SOMAttribute** pragma behave as though declarations of the _get and _set methods appeared in place of the data declaration. See “The SOMAttribute Pragma” on page 315 and “set and get Methods for Attribute Class Members” on page 290 for more information.

Note also that this third list contains all member functions and static data members, whether their access is public, protected, or private. This sometimes makes the compatibility rule overly constraining to a class designer, who may prefer to group the member function declarations logically or by access, or even to omit private methods from the class declaration provided to clients of the class. For this reason, VisualAge C++ provides a pragma that can be used to explicitly specify the release order for a class. If the **SOMReleaseOrder** pragma is used for a class, then the declaration order of member functions is no longer significant, and the compatibility rule is changed to require that new members be added at the end of the pragma.

```
// Original Class Definition:
#pragma SOMAsDefault(on) // define ensuing classes as SOM
class Bicycle {
  public:
     int Model;
     static int Count;
     Bicycle(); // defined elsewhere
     void showBicycle(); // defined elsewhere
#pragma SOMAttribute(Model,publicdata)
#pragma SOMReleaseOrder( \
   Model, \
   Count,\
   showBicycle()))
};
#pragma SOMAsDefault(pop) // resume prior setting of SOMAsDefault
```

In the revised version below, new methods and static data members are specified *after* the existing methods, within the **SOMReleaseOrder** pragma. Whether you place the declarations for the new methods and static data members before or after existing ones is not important, as long as you use **SOMReleaseOrder** to maintain the positions of existing functions in the release order:

```
// Revision:
#pragma SOMAsDefault(on)
class Bicycle {
  public:
     int Model;
     static int Count;
     static int NumberSold;
     Bicycle();
     void showBicycle();
     int sellBicycle(int); // defined elsewhere
#pragma SOMAttribute(Model,publicdata)
#pragma SOMReleaseOrder( \
   Model, \
   Count, \
   showBicycle()), \
   NumberSold, \
   sellBicycle(int))
};
#pragma SOMAsDefault(pop)
```

Note that in the example above, it is not necessary to specify the argument type (`int`) for `sellBicycle()`. If `sellBicycle()` were overloaded with multiple argument types (for example, `sellBicycle(int)` and `sellBicycle(int,char*)`), you would need to specify both overloads of the function in **`SOMReleaseOrder`**.

You can use the `/Fr` (give the release order of a class) option to have the compiler generate a **`#pragma SOMReleaseOrder`** for a class. For further details see "The SOMReleaseOrder Pragma" on page 333.

Default Release Order Rules

If you do not specify a release order for a class, the compiler orders methods (including the get and set methods of SOM attributes) in the order of their appearance within the class definition.

As long as you follow the guidelines given in this section (do not remove any public or protected methods or data members, and do not reorder previously released methods or static data members), you can provide new releases of your library and the programs that use that library will not need to be recompiled. Even if you are providing the library only to C++ programs and do not require SOM's ability to allow cross-language sharing of class and method definitions, this freedom from recompilation gives you more room to make minor adjustments or major enhancements to your library, and it decreases the resistance that those using the library might otherwise have to installing new versions of the library.

Upward Binary Compatibility of Libraries

Version Control for SOM Libraries and Programs

The release order of a class's data members, methods, and static member functions enables SOM client programs to work with new versions of SOM libraries without being recompiled. This means that a library can be recompiled after client programs have already been compiled and linked to an earlier version of the library. However, problems can occur if a program is compiled to one version of the library, and then a *lower* or backlevel version of the library is substituted. SOM implements a form of version control that can detect this situation.

The following scenario illustrates how version control works with SOM:

1. A SOM library containing a new version of the `Bicycle` class is compiled. The "version" of the class is major version 1, minor version 5 (or, for simplicity, version 1.5). This version is assigned within the class definition, using the **`SOMClassVersion`** pragma.
2. A program that uses the SOM library's definition of class `Bicycle` is then compiled. The compiler determines that the version of `Bicycle` the program was compiled to is version 1.5. The program runs successfully with this version of the library.
3. A new version of the SOM library becomes available, and class `Bicycle` is now at version 1.6. The program that was compiled to version 1.5 still works, because SOM libraries are upward compatible.
4. The program that uses the `Bicycle` class is copied to a different system, and class `Bicycle` in the SOM library on that system is at version 1.3.
5. When the program using `Bicycle` is loaded, the SOM runtime determines that a backlevel version of a `Bicycle` is being constructed, and it issues a warning message and ends the program. (If class version control were not used, the results of this run of the program would be unpredictable.)

SOM verifies that the major version is *the same* for a client and the objects it tries to create. When a SOM class increases its *major* version number, SOM assumes that an incompatible change has occurred.

You can use version control to ensure that programs do not experience unpredictable behavior as a result of using backlevel definitions of classes when more recent versions of those classes were expected.

Note: Currently the SOM runtime only tests for a compatible version of a class the first time an object of that class is instantiated. This can lead to problems in programs consisting of multiple compilation units, in which the uses of an object in one compilation unit expect a different version from the uses of that object in another compilation unit.

The following scenario illustrates the problem:

1. A program requests an instance of a SOM class `MyClass` at version 1 release 3. The SOM runtime determines that the current version of `MyClass` is version 1 release 4, so the object is created successfully.
2. Another compilation unit within the program requests an instance of `MyClass` at version 1 release 5 (because that compilation unit was compiled later than the first compilation unit). The SOM runtime does not check for version compatibility, because it already did so when the first `MyClass` instance was created. As a result, a program expecting at least version 1 release 5 of a class is given an object of an earlier (and possibly incompatible) version of that class.

If you update the version of a SOM class and recompile one of its clients, you should recompile all the clients of its class to avoid the problem described above.

Recompilation Requirements for SOM Programs

When you make changes to a SOM class, the type of change determines what parts of your program and its client code require recompilation. The following tables show the major types of changes you can make to a SOM class, and what code must be recompiled when you make any such change.

Notes:

1. Changing the signature or name of a method, or the name of a data member, or changing the access from private to protected/public or back, is equivalent to deleting one method or data member and adding another.
2. These tables list the access levels in the first column and the compilation units that need to be recompiled for adding, changing, and deleting elements in the second, third, and fourth columns, respectively. For example, for a private method, the entry under **Adding** is "Class, added method". This means that you have to recompile the compilation unit where the class is defined and, if it is a different compilation unit, the compilation unit where the new method is defined.
3. Classes that have all member functions declared inline are considered to be declarations according to the rules of C++. These "declarations" can appear in several different compilation units. If you change a member of such a class, the "class" entry in these tables means that you must recompile the compilation unit where the **`SOMBuildClass`** structures are created. See “The SOMDefine Pragma” on page 322 for more details.

Figure 28. Recompilation Required for Method Changes

Access	Adding	Changing the Implementation	Deleting
private	Class, added method	Class, changed method	Class
protected	Class, added method	Class, changed method	Class, friends, subclasses
public	Class, added method	Class, changed method	Class, friends, subclasses, all clients that referenced method

Figure 29. Recompilation Required for Data Member Changes

Access	Adding	Changing the Type	Deleting
private	Class, methods using new data, friends	Class, methods using changed data, friends	Class, methods that used data, friends
protected	Class, methods using new data, friends	Class, methods using changed data, all subclasses and friends	Class, methods that used data, all subclasses and friends
public	Class, methods using new data, friends	Class, methods using changed data, all subclasses and friends	Class, friends, subclasses, all clients that referenced the data

Note: Friends are assumed to have intimate knowledge of the implementation of a class. Because this knowledge includes knowledge of private data, friends are assumed to be created using the same language and compiler as the classes they are friends of, and they require recompilation whenever the class requires recompilation.

SOM and Interlanguage Sharing of Objects and Methods

You can share C++ classes with other programming languages either by using the **SOMAsDefault** pragma for those classes, or by deriving the classes from SOMObject. In either case, SOM restricts you from using certain C++ coding practices. These are documented in "Differences between SOM and C++" on page 294. This section outlines some of the issues you have to keep in mind if you want to share SOM objects with other languages. See the *SOMObjects Developer Toolkit Publications* for more details and for information on accessing SOM classes and methods from different programming languages. For more information on the individual SOM-related pragmas, see the descriptions in "Pragmas for Using SOM" on page 313.

SOM Requires a Default Constructor with No Arguments

One restriction SOM imposes that primarily affects interlanguage sharing of SOM objects, is the requirement that all classes have a default constructor that takes no arguments. In C++ you can declare a class with no default constructor:

```
class X {
  public:
    int Xdata;
    X(int a) {Xdata=a;};
};
```

When you compile a C++ client program that tries to call a nonexistent default constructor, VisualAge C++ issues a compile-time error, even when the SOM class the client is using was compiled separately. If you declare an X with the statement X b;, given the above class definition (regardless of whether or not it is a SOM class), the compiler issues an error. However, if the class is a SOM class, the compiler must anticipate potential calls to a nonexistent default constructor by SOM clients other than those compiled by VisualAge C++. Rather than generate an arbitrary default constructor (one whose behavior may or may not be the desired behavior for the class), the compiler generates one that results in a runtime error whenever it is called. Note that this behavior makes the class unusable with DSOM, which requires a valid default constructor.

In the following example, the defined class does not have a no-argument constructor. However, it has a constructor that has all default arguments:

```
class X {
  public:
    int Xdata;
    X(int a=3) {Xdata=a;};
};
```

VisualAge C++ generates two constructors for X if class X is a SOM class: a constructor that takes an integer argument whose value is assigned to Xdata, and a constructor that takes no argument and assigns the value 3 to Xdata.

Note that it is possible for client code written in another language to construct an object of a class that does not have a default constructor, provided the client code first calls SOMNewNoInit or SOMRenewNoInit for the object, and then invokes the constructor.

Accessing Special Member Functions from Other Languages

In C++ you can define an operator== for a class, then use the == operator to determine whether two objects of the class are equal. Not all languages support this concept of operator overloading. In order for programs not written in C++ to be able to access special member functions such as overloaded operators, you must provide names with which these functions can be called from non-C++ programs. The compiler uses these names to generate appropriate IDL definitions for these operators. You can rename class operators using the **SOMMethodName** pragma, described on page

328. The following class definition provides SOM names through which non-C++ programs can access the operators of the class:

```
#include <som.hh>
class Bicycle:  public SOMObject {
  public:
     int model;
     Bicycle();
     int operator==(Bicycle& const b) const;
     int operator <(Bicycle& const b) const;
     int operator >(Bicycle& const b) const;
     Bicycle& operator =(Bicycle& const b);
#pragma SOMMethodName(operator==(),"BicycleEquality")
#pragma SOMMethodName(operator <(),"BicycleLessThan")
#pragma SOMMethodName(operator >(),"BicycleGreaterThan")
#pragma SOMMethodName(operator=(),"BicycleAssign")
};
```

Non-C++ programs can then call these special member functions by referring to their SOM names (`BicycleEquality` etc.).

Assignment Methods

The compiler provides four SOM assignment methods for a SOM class by default, one of which is called when the compiler encounters an assignment operator. If you define an `operator=` for a class, the compiler does not generate any assignment methods, in which case calls using the SOM method names will call the appropriate user-defined assignment operator.

The SOM assignment methods have the following SOM names and prototypes:

```
SOMObject *somDefaultAssign(somAssignCtrl *, SOMObject *)
SOMObject *somDefaultConstAssign(somAssignCtrl *, SOMObject *)
SOMObject *somDefaultVAssign(somAssignCtrl *, SOMObject *)
SOMObject *somDefaultConstVAssign(somAssignCtrl *, SOMObject *)
```

The `somAssignCtrl` parameter allows SOM to handle base class assignment to ensure that each base is only assigned once when a base class appears multiple times in an inheritance hierarchy. A user-defined `operator=` method does not give you this capability. Therefore, if you code your own assignment method in a class that has multiple parents (not including SOMObject), you should use the SOM assignment methods rather than `operator=` to ensure correct results. Note that, except when an `operator=` method is defined, the compiler generates SOM assignment methods for any that are not user-defined.

You should place any user-defined assignment methods (`operator=`) in the release order for the class. You do not need to put compiler-defined assignment methods into the release order unless you want to take their address. Do not put the SOM assignment methods in the release order, because they are introduced in SOMObject.

If you want to define a class that can be used by a client either as a C++ class or as a SOM class using the SOM assignment methods, define both the `operator=` functions and the SOM assignment methods, using conditional compilation to determine which are included in the class definition.

All operators you provide for a class, except for the default assignment operator, must be given SOM names using the **SOMMethodName** pragma, if you want them to be easily callable from non-C++ programs. Otherwise, their names will be "mangled" by the compiler. This includes the **new** and **delete** operators, if you define them at the class level. You need to specify a SOM name for non-default constructors, because they are overloaded versions of the default constructor. You do not need to specify a SOM name for the default constructor or the destructor (the compiler automatically gives these functions the names **somDefaultInit** and **somDestruct**).

Invoking Constructors from Other Languages

Given a default constructor of the form:

```
ClassName();
```

VisualAge C++ generates a function with the following signature for use by non-C++ programs:

```
void somDefaultInit(this, Environment*, InitVector*);
```

The non-C++ program must ensure that the vector pointer and the environment pointer are correctly set or are NULL. (You should always use a NULL value; the compiler may use a non-NULL value in some cases, but user code that passes a non-NULL value will behave unpredictably.) The presence or absence of the environment pointer is dictated by the callstyle of the class. (See “IDL and OIDL Callstyles” on page 293 for further details.) The bindings generated by the SOM compiler normally ensure that the pointers are correctly set or are NULL.

Copy constructors have one of the following names generated for them:

```
   somDefaultCopyInit
   somDefaultConstCopyInit
   somDefaultVCopyInit
   somDefaultConstVCopyInit
```

Other nondefault constructors are given a mangled name unless you supply a SOM name using the **SOMMethodName** pragma.

When invoking a nondefault constructor from outside of C++, you should first create the object using **SOMNewNoInit** or **SOMRenewNoInit**, and then invoke the constructor.

If you use **SOMNew** or **SOMRenew** and then invoke the constructor, you will end up initializing the same object twice.

set and get Methods for Attribute Class Members

SOM supports two types of data members: attributes and instance variables. Depending upon the pragma setting, the compiler generates default get and set methods for these attributes if you do not supply your own. If you specify **#pragma SOMAttribute(readonly)** for an attribute, no set method is generated or definable. An attribute is a nonstatic data member for which you have specified **#pragma SOMAttribute**. SOM predefines methods to set and get the value of attributes. Attributes have the following properties:

- Attributes are the only way of accessing data in classes used in DSOM applications.

  If you fail to declare an attribute and attempt to directly access instance data in a remote object, you will receive runtime error 20109 from SOM, and a message resembling the following:

  ```
  somDataResolve error: class <X_Proxy> is abstract with respect to <X>
  ```

- Attributes allow the class implementor to add instrumentation or other side effects to data access by explicitly defining the `_get` and `_set` methods with the desired function.
- You do not need to define methods to set or get the value of an attribute. This is done automatically by the compiler. You can override these methods where the automatically defined method does not provide the required functionality.
- The names of the set and get methods are consistent and predictable: for an attribute `j`, the methods are `_set_j()` and `_get_j()`. (For C++ programs using the attributes, you can get or set the attributes using the attribute names rather than the get and set methods.)
- You can identify whether the compiler should automatically generate get or set methods for an attribute, or whether to use a user-defined get or set method.

Get and set methods have the following signatures for scalars, arrays, and structs/unions/classes:

```
// when 'indirect' attribute is not used with SOMAttribute pragma:
T _get_var() const volatile;      // scalar var of type T - get
void _set_var(T) volatile;        // scalar var of type T - set


T& _get_var() const volatile;     // scalar var of type T - get, when
                                  // SOMAttribute(...,indirect) is specified
void _set_var(const volatile T&) volatile;
                                  // scalar var of type T - set, when
                                  // SOMAttribute(...,indirect) is specified

T* _get_var() const volatile;     // arrays of var of type T - get
void _set_var(const volatile T*) volatile;
                                  // arrays of var of type T - set

T _get_var() const volatile;      // structs/unions/classes of type T - get
void _set_var(const volatile T&) volatile;
                                  // structs/unions/classes of type T - set
```

Note that pointers are used rather than references, for arrays of `T`. This is done because the interface treats the type as a pointer to the first array element rather than as a pointer to the entire array.

You do not need to declare the get and set methods for an attribute in your class declaration, if you choose to have the compiler automatically generate them for you. The compiler treats the get and set methods for an attribute as being declared whether it encounters a declaration or not. The **SOMAttribute** pragma determines whether the get and set methods are *defined* by the compiler, provided by the programmer, or, in the case of the set method, not provided at all. If the **SOMAttribute** is not used, attributes are not created.

See “The SOMAttribute Pragma” on page 315 for further information on attributes.

Interface Definition Language (IDL) Considerations

The Interface Definition Language (IDL) is a facility for defining the interface of SOM classes. IDL provides a CORBA-compliant description of a SOM class. When you compile a SOM-enabled C++ program with VisualAge C++, the compiler can generate IDL definitions for SOM classes the program defines. If you are writing code in another language and you want to create objects of those SOM classes, you normally use an **.idl** file to generate a header file for your program so that the SOM classes you use are visible to the compiler in question. The **sc** translator uses the **.idl** file to generate the necessary bindings for the other language, and also to load the Interface Repository (IR), which is used by DSOM.

Interface Definition Language

If you are creating SOM classes and you anticipate that all users of your classes will be coding only in C++, you do not need to consider the impact of IDL on how you code and on what pragmas you use. However, if there is any likelihood of non-C++ programs using your SOM classes, you need to understand the connections between IDL and VisualAge C++.

If you are writing code to work with an existing SOM interface, you may start out with IDL interfaces. You can use the SOM compiler from the SOMObjects Developer Toolkit if to create a C++ **.hh** file from the IDL definitions.

The remainder of this section explains those connections.

Generating IDL for C++ SOM Classes

To generate IDL for a C++ SOM class, you should first ensure that the SOM class is declared in a **.hh** file (as opposed to the usual **.h** file used for C++ class declarations). This **.hh** file can be included by C++ source files that use the class, just as **.h** files can. When you want to generate the IDL for a class, compile the **.hh** file itself, rather than the C++ source files that include it. The compiler will produce a **.IDL** file containing the class IDL definition. You do not need to specify any SOM-related options for the IDL to be generated.

You can use **#pragma SOMIDLTypes** within your **.hh** files to group types together. See "The SOMIDLTypes Pragma" on page 325 for further details.

IDL Types and C++ Types

IDL names for the following built-in C++ types are identical to those types' C++ names:

- **short**, **unsigned short**, **long**
- **float**, **double**
- **char**.

The following C++ types are mapped to the IDL types indicated:

- **signed char** is mapped to **octet**
- **unsigned char** is mapped to **char**
- **int** is mapped to **long**
- **long double** is mapped to **double**
- **unsigned int** is mapped to **unsigned long**
- **wchar_t** maps to **unsigned short**
- **char*** maps to **string** when it is a parameter, otherwise it maps to **char***
- Enumerated types are mapped to integer constants.

IDL Names and C++ SOM Pragmas

If you do not use any of the SOM pragmas **SOMMethodName**, **SOMClassName**, or **SOMNoMangling**, the names of SOM class methods and class templates are mangled by VisualAge C++. These mangled names are the names that appear in the program's **.idl** file, and these names are likely to be long and difficult to understand. Although you can access SOM classes and their methods using these mangled names, this practice is error-prone and unnecessarily complicated. You can use the above pragmas to make the SOM names for your classes more understandable.

IDL requires that class and method names be distinct and case-insensitive. VisualAge C++ normally ensures this by mangling class and method names. Mangling encodes case differences, and also reflects argument types of overloaded methods in their SOM names.

If you use the **SOMClassName** pragma to attach a SOM name to a class, make sure that the name you select is unique without regard to case. If you use the **SOMNoMangling** pragma for a class or a range of classes, method names in those classes are not mangled, which creates conflicts between any names that differ only in case, and between different overloads of functions. You can use the **SOMMethodName** pragma to correct this situation, by associating SOM names with individual methods.

- IDL matches methods by their names only. It does not support method overloading. This means that you must differentiate overloaded methods of a class by using the **SOMMethodName** pragma on overloaded methods.
- IDL is case-insensitive. If you define a C++ method `print` to print an object, and a C++ method of the same class called `prInt` to print an integer data member of that object, their IDL names will be the same if you use the **SOMNoMangling** pragma, unless you rename one of the methods using the **SOMMethodName** pragma.
- If you use the **SOMNoMangling** pragma for a class or a range of classes, method names in those classes are not mangled. This can result in multiple overloaded functions mapping to the same name. The compiler detects such conflicts and issues an error message. You can use **SOMMethodName** to resolve these conflicts.
- Changing the IDL name of a method can break binary compatibility because IDL matches methods by name only.

IDL and OIDL Callstyles

The Common Object Request Broker Architecture (CORBA) defines an implied second parameter of type **Environment*** for SOM methods and static member functions. This parameter can be used to pass extra information between SOM methods and clients, such as exception information indicating that a SOM method could not be called. In initial releases, SOM did not support this second parameter. This can result in compatibility problems because new code may have the extra parameter while old code, including such classes as **SOMObject** and **SOMClass**, may not. The presence or absence of this second parameter in a class method or static member function is referred to as the method or function's *callstyle*. The new callstyle with the **Environment*** parameter is referred to as the IDL callstyle, while the old callstyle without that parameter is referred to as the OIDL callstyle (for “Old IDL”).

To preserve binary compatibility with old SOM application code, SOM now supports both callstyles. This leads to a model where some methods in a program may expect environment pointers, while others may not.

The callstyle is determined on a class-by-class basis. For a given class, either all methods *introduced by that class* will expect an environment parameter, or none will.

Note: The callstyle of an inherited method is the callstyle of the class in which the method is defined, not the callstyle of the inheriting class.

You can specify the callstyle for a class using the **SOMCallStyle** pragma. By default, all classes will have the IDL callstyle.

Callstyles and Pointer-to-Member

You cannot assign the address of an IDL-callstyle method to a pointer to an OIDL-callstyle method, or vice versa. Whether a pointer to member is an IDL- or OIDL-callstyle pointer depends on the class the pointer to member is declared in. If the declaring class uses IDL callstyle, the pointer to member can only point to IDL-callstyle methods; otherwise it can only point to OIDL-callstyle methods. Note that conflicts between callstyles are unlikely to occur, because IDL is the default callstyle.

C++ Limitations to IDL

When IDL is generated for a C++ class, the bodies of inline functions are not emitted in the IDL. As a result, if you later translate the IDL file back to a C++ header file, inline function definitions become function declarations with no function body.

VisualAge C++ does not support inlining of C++ member functions when IDL is generated, and all member functions of SOM classes are called out-of-line.. Because inlining may be supported in the future, you should consider the bodies of public inline functions to be a part of the public interface of a class if you are concerned about upward binary compatibility of your classes.

Differences between SOM and C++

SOM imposes a slightly different view of object orientation on its classes than does C++. This section describes differences between the object-oriented features of C++ and those supported by SOM.

Initializer Lists and Constructors

You cannot use an initializer list to initialize an object of a SOM class, because all SOM classes have constructors, and C++ language rules do not allow classes with constructors to be initialized in this way.

Function Overloading

C++ lets you define multiple methods within a class that have the same name, but different combinations of arguments. These arguments are collectively known as a method's *signature*, and a class that defines multiple instances of a method with different signatures is said to overload that method. A class can overload static member functions as well as methods.

SOM does not support the C++ concept of function overloading, either for methods or for static member functions. By default VisualAge C++ generates mangled names for all overloaded functions so that different overloads can be distinguished. If both your SOM classes and the programs that use them are coded in C++, you can easily overload functions because the compiler uses this consistent name-mangling scheme to resolve overloaded calls. However, if you plan to make your SOM classes accessible to programs written in languages other than C++, you should not rely on C++ name mangling, because the mangled names are often difficult to understand. Instead, you should provide SOM with a function name to call for each signature of an overloaded function. You do this using the **SOMMethodName** pragma. The following example shows three declarations of method `add()` for a class, and three **SOMMethodName** pragmas that make all three methods clearly accessible to SOM programs written in other languages:

```
class Bicycle : public SOMObject {
  public:
     // ...
     void add(Bicycle& const);
     void add(int);
     void add();
#pragma SOMMethodName(add(Bicycle& const),"AddBike")
#pragma SOMMethodName(add(int),"AddInt")
#pragma SOMMethodName(add(),"AddVoid")
};
```

You could avoid the above **SOMMethodName** pragmas by relying on the C++ mangling scheme, but this would make client code more difficult to write or maintain. For example, the following function in C++:

```
x::operator=(const volatile x);
```

is mangled to the following:

```
dts____as__frxzvx
```

For classes in which the **SOMNoMangling** pragma is in effect, you must use the **SOMMethodName** pragma for all but one of the overloaded versions of a given method or static function. For the sake of code clarity you should use the **SOMMethodName** pragma to rename *all* signatures of a function that is overloaded.

Calling Methods Through a NULL Pointer

Some implementations of C++ allow you to call nonvirtual functions through a NULL pointer. You cannot do this in SOM-enabled C++ programs. If you call a nonvirtual function through a NULL pointer in a SOM-enabled C++ program, the program may compile successfully but it will not run correctly. For example, the call to the virtual function `vf()` below causes a trap in both native C++ and SOM-enabled C++, while the call to the nonvirtual function `nvf()` causes a trap only in SOM-enabled C++:

```
class A {
   public:
       void nvf();
       virtual void vf();
} *a = NULL;

void hoo(){
     a->nvf();    // OK in C++, traps in DTS C++
     a->vf();     // Traps in both because virtual.
}
```

Data Member Offsets

C++ lets you determine the offset of data members into an object. An expression such as:

```
int ((char*)&Instance.Member - (char*)&Instance);
```

can be used in C++ to determine how far into an instance `Instance` the member `Member` is located. This syntax is also supported in SOM. However, the result of the expression may not be identical for subclasses. Given:

```
class Base : public SOMObject { public: int i; } B;
class Derived : public Base { /* ... */ } D;
#define MyOffset(Obj,Member) int((char*)&Obj.Member - (char*)&Obj)
```

the equality `MyOffset(B,i) == MyOffset(D,i)` may or may not hold, depending on how SOM determines the data reordering scheme for each class.

The offsets of data members into an object are contiguous within each access-specifier (**`public`**, **`protected`** or **`private`**), and are assigned to each block in the order of declaration.

Casting to Pointer-to-SOM-Object

The structure of SOM objects requires that the memory layout of the instance begin with a pointer to an appropriate method table. This differs from normal C++ objects in which no such pointer is allocated unless the class has virtual functions. The result of this difference is that it is not generally possible to treat arbitrary storage as a SOM object. In particular, casting 0 to a pointer to a SOM object is not recommended. You can get unexpected results when a SOM pointer is cast to a non-SOM pointer. See "Determining which new and delete Operators Are Used" on page 306 for an example of such unexpected results.

Dereferencing a Virtual Base Pointer to a Derived Base

In native C++ a pointer to virtual base cannot be explicitly cast to a derived base. This is allowed in SOM-enabled C++. The following example illustrates this difference between native and SOM-enabled C++:

```
#include <som.hh>

struct vbstruct : public virtual SOMObject {
#pragma SOMDefine(*)
};

void main() {
   SOMObject *p = new vbstruct; // always legal
   vbstruct *q;
   q = (vbstruct *) p;          // legal for SOM, not for non-SOM
   q = p;                       // always illegal (need a cast)
}
```

Differences between SOM and C++

Multiple Inheritance of a Base Class

SOM does not implement multiple occurrences of the same nonvirtual base. For example:

```
#ifdef __SOM_ENABLED__
class A : public SOMObject { /* ... */ };
#else
class A { /* ... */ };
#endif
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class MyClass : public B, C { /* ... */ };
```

The compiler issues an error for the definition of class `MyClass` if class `A` is a SOM class. If class `A` is not a SOM class, the program compiles without an error.

Note: The compiler cannot warn you about multiple inheritance errors in SOM programs when different classes in an inheritance graph are separately changed and recompiled. In the following example, assume that each struct is declared in a separate file and compiled on its own:

```
struct s     {};
struct a:s   {}; // based on s
struct b     {};
struct d:a,b {}; // based on a, b, and s
```

If the file containing `struct b` is changed to:

```
struct b:s   {}; // based on s
```

and recompiled individually, the compiler will not warn you of the error, and programs using struct `d` may behave unpredictably.

Local Classes

Local, non-file-scope classes may not be SOM classes. However, a local, non-file-scope class may have a nested class that is a SOM class. In the following example the declaration of class `CantBeFromSOM` causes a compiler error because it only has the scope of `main`:

```
class IsFromSOM: SOMObject { /* ... */ };
void main() {
   class IsntFromSOM { /* ... */ };
   class CantBeFromSOM: SOMObject { /* ... */ };
   }
```

Abstract Classes

An *abstract* class is a class with one or more pure virtual functions. Abstract C++/SOM classes are supported. If the abstract class does not define a default constructor, VisualAge C++ prevents calls to the constructor from other C++ programs. The IDL generated by VisualAge C++ also prevents calls to a nonexsistent abstract class constructor from programs written in other languages.

As usual with C++, you can provide your own method bodies for pure virtual member functions. If you do this the method bodies must be provided in the same file as the definition of the first member that is not inline, or in the same file as a **SOMDefine** directive.

Classes as Objects

In native C++, a class is a syntactic entity that exists only at compile time; it has no representation outside of the source code that defines it. A C++ class cannot be an object, and a C++ object cannot be a class. The strict distinction between classes and objects does not hold for SOM. A SOM class always exists at runtime, and is itself a SOM object.

Because SOM classes are runtime objects, they can provide a number of services to client objects. For example, a SOM class can respond to specific inquiries regarding the interface of its instances; each SOM class includes a method named **somSupportsMethod**, which when invoked with any string returns a Boolean value indicating whether the string represents a method supported by instances of the class. SOM class objects can also provide information to clients such as its name, the names of its base classes, the size of its instances, the number of methods it supports, and whether a provided SOM object is an instance of the class.

The SOMObjects Toolkit documentation describes a method for extracting the class object of a class, where an object of that class already exists. For example, you can call `obj->somGetClass()`, to extract the class object for object `obj`.

Where you need to name the class object but you do not have an instance of it, the Toolkit allows you to code the class name, preceded by an underscore. For example:

```
SOMObject* anotherObj;
anotherObj->somIsInstanceOf(_Foo); // toolkit syntax
```

This syntax is not supported with DTS classes, because it imposes on the user's identifier space as defined by ANSI. Instead, VisualAge C++ introduces a static member to each class it converts to a SOM class:

```
SOMClass * const __ClassObject
```

This static member cannot be added to the release order for the class. You can use the following syntax in place of the toolkit syntax shown above, for DTS classes:

```
   anotherObj->somIsInstanceOf(Foo::_ClassObject);
```

Although you can refer to this member as *className*::_ClassObject from within a C++ program, it is not a "real" data member in that it does not exist in memory. The compiler resolves references to this member to a pointer to the class object for *className*.

Metaclasses

A SOM class is also an instance of a class, because all SOM classes are objects. A class whose instances are other classes is called a metaclass. A metaclass definition specifies the interface of a class, just as a class definition specifies the interface of an object. The SOM metaclass has no conceptual equivalent in C++. The SOM metaclass exists at runtime, is capable of providing specific services to client code, and may be used as a parent of other metaclasses. For more details on the concept of metaclasses, see the online *SOM Programming Guide*.

When you create a class in SOM, the appropriate metaclass is created automatically if you do not specify one. You can also explicitly create your own metaclasses. You can create a metaclass by deriving from **SOMClass**, so that your metaclass can perform functions such as keeping track of what SOM classes have been constructed in a program. (A **SOMClass** object is constructed for every SOM class used by a program, the first time an object of that class is constructed.) To create a metaclass, follow these steps:

1. Derive a new class from **SOMClass**, which is declared in som.hh.
2. Associate this new class with the instance class via the **SOMMetaClass** pragma.

For example:

```
#include <som.hh>

class MyMeta : public SOMClass { /* ... */ };
class MyClass : public SOMObject {
   // ...
   #pragma SOMMetaClass(*,MyMeta)
};
```

Note: The compiler does not distinguish between metaclasses and other classes. For SOM to function correctly, you should derive all metaclasses from **SOMClass**.

offsetof macro

The **offsetof** macro does not work as well with SOM classes as it does with regular C++ classes. Its value is determined at runtime, as the relative positioning of the data “blocks” introduced by each base are not known until then. This means that **offsetof** is not a reliable way to determine the position of a member within a subclass. The value of the **offsetof** macro for a member of a base cannot be assumed to be correct for subclasses of that base.

sizeof operator

The **sizeof** operator works differently for **SOM** objects than for non-SOM objects. The **sizeof** operator indicates the size in bytes of the object it is applied to. For non-SOM objects, this size is determined at compile time, and can therefore be used in expressions evaluated at compile time. For SOM objects, **sizeof** returns a value that is determined at runtime. This means that you cannot apply the **sizeof** operator to SOM objects in situations where the value must be determinable at compile time, such as array bounds (for static initializers), case expressions, bit field lengths, and enumerator initializers. For example, the following uses of **sizeof** will cause compilation errors:

```
class MyClass {
   public:
      int i:sizeof(Buffer);
};
enum { E = sizeof(MyClass) } x;
try   Buffer myBuffer[sizeof(Buffer)]; // Buffer is a SOM class
switch(/* ... */) {
   case sizeof(Buffer): break;
}
```

Instance Data

SOM supports both static data members and arrays. An array of SOM objects is represented as a pointer to an array of SOM object instances.

Templates

You instantiate a template class as with native C++. If you want to avoid compiler mangling of template names, you should also supply a SOM name for any instantiation of a template class. For example:

```
typedef Stack<int> IntStack;
#pragma SOMClassName(Stack<int>, "IntStack")
IntStack MyIntStack;
```

This declares an object `MyIntStack` of type `Stack<int>`. This could also be coded as:

```
Stack<int> MyIntStack;
#pragma SOMClassName(Stack<int>, "IntStack")
```

You can achieve the same effect by coding:

```
#pragma define(Stack<int>) // instantiates class Stack<int> from template
#pragma SOMClassName(Stack<int>, "IntStack")
```

Note that the first argument of the **SOMClassName** pragma (the class to be renamed) must be the template class with its type argument, rather than the typedef.

If you plan to make a template class accessible to non-C++ programs, you must define an implementation of the template class for each type that will be requested by those programs. You can do this either with the **SOMDefine** pragma, or by instantiating the template within the C++ program. For example:

```
typedef Stack<int>    IntStack;    // assume Stack is a SOM class
typedef Stack<double> DoubleStack; // template
typedef Stack<char>   CharStack;
typedef Stack<float>  FloatStack;
// ...
IntStack i;                        // makes IntStack available
                                   // to non-C++ programs
#pragma SOMDefine(Stack<double>)   // makes DoubleStack available
#pragma SOMDefine(CharStack)       // makes CharStack available
                                   // FloatStack is not available
```

You should then use the **SOMClassName** pragma to provide SOM names to the template instantiations, so that the compiler does not generate mangled names for those instantiations.

When using templates to implement SOM classes, do not include information dependent upon the instantiation type within the class description. For example, the following code produces a runtime error because the `SOMAttribute` pragma is processed for both implementations, and each one is incorrect for the other implementation:

```
#include <som.hh>

template <class T, int S = 5>           // default arg value
class D : public SOMObject {
  public:
      T Velocity;
  #pragma SOMAttribute(D<int>::Velocity, readonly)
  #pragma SOMAttribute(D<int, 9>::Velocity, readonly)
};

#pragma define(D<int>)
#pragma define(D<int, 9>)
```

Instead, use a single `SOMAttribute` pragma for each attribute within a template class. For the above example, the pragma would appear as:

```
   #pragma SOMAttribute(Velocity, readonly)
```

In cases within the class description where a class name is expected, such as the **SOMNoMangling** or **SOMNoDataDirect** pragmas, you should use an asterisk (*) for the class name.

Methods of Template Classes

Methods of a template can be renamed using the **SOMMethodName** pragma. You do not need to rename template methods, but if you plan to make your SOM classes available to non-C++ programs, you can make the interface to your classes simpler by renaming methods. If you do not rename template methods, the compiler mangles their names, and the mangled names are difficult to remember and are likely to lead to typographical errors.

You should use the **SOMMethodName** pragma to rename the methods of a template class for each type you plan to instantiate the template with from a non-C++ program. For example, if you define a template class:

```
template class <T> class MyTemplate {
   public:
      T dataMember;
      void Push(T item);
   };
```

and you anticipate your template being used with types **int** and **double**, you should add pragmas such as the following to the C++ program:

```
#pragma SOMMethodName(MyTemplate<int>::Push(int),"PushInt")
#pragma SOMMethodName(MyTemplate<double>::Push(double),"PushDouble")
```

Memory Management

This section describes how memory is allocated to SOM objects, and tells you how to use the **new** and **delete** operators for memory allocation.

Heap and Stack Memory Allocation

C++ programs can store objects in two different areas of memory, known as the stack and the heap. The stack and the heap are implemented by software. They are distinguished by the fact that objects stored on the stack are automatically deleted when the function or block within which they were created passes out of scope, while objects stored on the heap must be explicitly deleted.

Objects allocated with the **new** operator are placed on the heap, including SOM objects. Automatic objects are usually allocated in the current stack frame. SOM objects that are declared as having automatic duration, rather than as pointers to objects, are usually allocated on the current stack frame. As with normal C++, the **new** operator is not called for automatic duration operators.

Differences between SOM and C++

Overloading the new and delete Operators

You can overload the **new** and **delete** operators either on a class-specific basis or globally. Because most programs will contain a mixture of SOM and non-SOM objects, the compiler provides two different paths for memory allocation and deallocation using **new** and **delete**, one for SOM objects and one for non-SOM objects.

You can have multiple, distinguished versions of operator **new** within a class. The operator **delete** is restricted to one version per class.

SOM accepts an additional parameter to an operator **new** for a SOM class, which points to the class's class object. An operator **new** for a SOM class has one of the following forms:

```
void *operator new (size_t InstanceSize);
void *operator new (SOMClass* ObjClass, size_t InstanceSize);
```

The SOM version of the global operator **new** has the form:

```
void *operator ::new (SOMClass* ObjClass, size_t InstanceSize);
```

You can use the `SOMClass*` parameter, in both class and global definitions of operator **new**, to have a pointer to the object's class object passed to the operator. For any class that is a SOM class, the compiler passes this parameter whether you specify it in the operator's declaration or not. You do not specify this argument when invoking **new**, so there is no way for a call to **new** to specify its own value for the `SOMClass*` argument.

You cannot have both types of operator **new** within a class. You can have both types of global operator **new**. Note that even if you use placement arguments in an operator **new**, the `SOMClass` argument is always the first argument.

The `SOMClass*` argument appears first because this allows the compiler to differentiate between a SOM operator **new** and a non-SOM operator **new** that takes a `SOMClass*` as an argument. You can use the `SOMClass*` argument, for example, to print the class name, by calling `thisClass->somGetName()` where `thisClass` is a pointer to a SOM class.

The **delete** operator for SOM classes has the same form as for other C++ classes. For a given class, you can have at most one of the following forms of operator **delete**:

```
void operator delete(void*);
void operator delete(void*, size_t);
void operator delete(SOMObject*, size_t);
```

For the sake of easily maintained code, you should always include the **size_t** argument, whether you use it or not, because it allows you to later change to an implementation that does use the argument, without requiring client programs to be recompiled.

The first argument is a pointer to the object instance being deleted. Because of the way that SOM uninitializes an instance, the first word of the object still points to the object's method table, which in turn points to the class object. This gives you access to information about the specific class being deleted.

You can also code a SOM version of the global **delete** operator, of the form:

```
void operator ::delete(SOMObject*, size_t);
```

The type of the first argument is SOMObject to distinguish the function signature from the non-SOM global **delete** operator. Note that the compiler recognizes such a replacement based on the exact signature. You must include both arguments in the declaration.

By default, this function calls SOMFree to deallocate the SOM object's storage.

The following example shows how you can define **new** and **delete** operators for a SOM class. In the example, the **new** operator increments a counter each time it is called, and then calls the global **new** operator to allocate storage for the object. The **delete** operator decrements the same counter, and then calls the global **delete** operator to deallocate the storage. The counter is a static class member that can be accessed to determine how many objects of the class currently have storage allocated to them by **new**.

```
#include <som.hh>
class A : public SOMObject {
  public:  void* operator new(SOMClass*, size_t);
           void  operator delete(SOMObject*);

           static int howMany; // # of dynamically alloc instances
};

int A:howMany;

void* A:operator new(SOMClass *cls, size_t sz)
{
    howMany++;
    return :operator new(cls, sz);
}

void A:operator delete(SOMObject* obj)
{
    howMany--;
    :operator delete(obj);
}
```

Using new.h in C++ SOM Programs

If you normally include **new.h** in a program to specify that previously allocated storage is to be used when new is invoked, you should include **somnew.h** instead if the classes that make use of **new** are SOM classes.

Determining which new and delete Operators Are Used

If a SOM class has an operator **new** or an operator **delete**, these operators are used for all invocations of **new** or **delete** regardless of their signatures. If a SOM class does not have an operator **new** or an operator **delete**, the SOM version of the global operator is used.

Warning: Memory allocated by SOMMalloc can only be freed by SOMFree, and memory allocated by **malloc** can only be freed by **free**. If you use the SOM function for allocating storage for an object, and the non-SOM version for deallocating it (or vice versa), a runtime exception may occur.

For example, the following will cause a runtime exception:

```
class A : public SOMObject {
   public:
      operator delete(void* o, size_t s) { ::delete o; )
};
```

because class A's **delete** operator will be invoked when an object of class A is deleted. The first parameter will point to the object to be deleted. Note that because the first parameter is declared to be of type void*, this invocation implicitly involves converting a SOM pointer (an A*) into a non-SOM pointer (a void*). The subsequent `::delete o` therefore uses the global non-SOM **`delete`** operator, which calls **`free`**, instead of the global SOM **`delete`** operator that calls SOMFree.

Volatile Objects

The SOM class member functions are not defined to operate on volatile SOM objects. If you want to use the `volatile` qualifier with SOM objects, you must supply volatile versions of the SOM class member functions. In particular, you must supply volatile versions of the four compiler-supplied `operator=` functions (described in "Accessing Special Member Functions from Other Languages" on page 287). Note that if you supply a `const volatile` version of a function, you should also supply a `const` version of the function for the sake of runtime efficiency.

Data Members Implemented as Attributes

You cannot take the address of a data member that is implemented as an attribute.

If an attribute is made virtual by the `SOMAttribute` pragma, it will no longer behave like a normal C++ data member. Because attributes are accessed using get and set methods, making an attribute virtual in fact makes the get and set methods for the attribute virtual, and such virtual methods can be overridden in a derived class. A derived class that overrides these methods can therefore change the type or other characteristics of the data. This differs from normal C++ behavior in which a derived class cannot override definitions for data members defined in a base class.

SOM methods are all implicitly given `system` linkage. If the address is taken of a static member function, the resulting pointer value will be a pointer to a function with `system` linkage. The resulting pointer can only be assigned to a function pointer that also has `system` linkage.

Converting C++ Programs to SOM Using SOMAsDefault

The easiest way to convert existing classes to SOM classes is to use the **`SOMAsDefault`** pragma or the `/Ga` (enable implicit SOM mode) compiler option to tell the compiler what classes to treat as SOM classes. Both the pragma and the option include the required SOM header file `som.hh`, and implicitly convert all classes to SOM classes until implicit mode is turned off by a subsequent **`SOMAsDefault`** pragma.

When you implicitly derive classes from SOMObject, the compiler is said to have "implicit SOM mode" turned on.

If your programs do not use any of the C++ features that are not supported by SOM (such as multiple virtual inheritance), you should be able to compile and run them without further change. See "Differences between SOM and C++" on page 294 for information on C++ features that are not supported or are implemented differently for SOM programs.

VisualAge C++ converts all structs and C++ classes to SOM classes unless the files in which they are defined are in directories excluded from conversion to SOM by the `/Xs` compiler option. Files in any directory specified by the `/Xs` option (as well as certain standard directories of files to exclude) are not converted into SOM classes. See the *User's Guide* for further details.

Unions cannot be SOM classes.

Non-virtual multiple inheritance is not allowed. Suppose that a class `A` has the class `B` in at least two separate places in its class hierarchy. If class `B` is not a virtual base class, class `A` cannot be a SOM class.

Note: Member functions of implicit SOM classes are given `SYSTEM` linkage. This means that pointers that are supposed to point at static member functions of such classes must be explicitly declared `SYSTEM`.

Creating SOM-Compliant Programs by Inheriting from SOMObject

To make your programs SOM-enabled using this technique, you must first include the following header file in your program, before the first occurrence of a SOM class:

```
#include <som.hh>
```

Then, if you want to define a class that is SOM-enabled, you must inherit it from SOMObject, or from a class that itself was inherited from SOMObject. Note that all classes in a class hierarchy must be SOM classes, if any is a SOM class.

```
#include <som.hh>
class MyClass : SOMObject { /* ... */ };  // both these classes
class SubClass : MyClass { /* ... */ };   // are SOM-enabled

class EnclosingClass { SubClass a; };     // NOT SOM-enabled
```

Note that SOMObject has the special property of always being virtual.

Creating Shared Class Libraries with SOM

When you create a shared class library that contains SOM-enabled classes, you must export the following three symbols for each SOM-enabled class, in order to be able to use that class:

- *SOMClassName*`ClassData`
- *SOMClassName*`CClassData`
- *SOMClassName*`NewCLass`

You do this by adding each name to its own line in an exports file.

`DLLs` that are to be dynamically loaded using methods supported by the SOM Class Manager, such as SOMClassMgr::somFindClsInFile(), should also export an entry point called SOMInitModule that calls the compiler-defined NewClass function for each class defined in the DLL. For a DLL defining a single class whose SOM name is *SOMX*, this entry point could be written as:

```
void _Export _System SOMInitModule(long, long, char*)
{
  SOMXNewClass(SOMX_MajorVersion, SOMX_MinorVersion);
}
```

For more information about SOMInitModule, see the *SOMObjects Developer Toolkit Publications*.

Using SOM Classes in DSOM Applications

Distributed SOM, or DSOM, allows remote objects to appear local to a client program. Remote objects are implemented "under the covers" by the DSOM runtime. Remote objects are dynamically subclassed, and must always be treated as possible subclasses. This means that you must handle DSOM objects using pointer notation.

You cannot create or delete DSOM objects with the **new** and **delete** operators described in this book. For methods of creating DSOM objects, see the *SOMObjects Developer Toolkit Publications*.

You cannot access data directly for a DSOM object, because the object may reside on a different system. You must use the get and set methods instead. This means you must use the `SOMAttribute` pragma for all data you want to make accessible through DSOM. For a SOM class to be usable as a DSOM class, **`#pragma SOMNoDataDirect(on)`** must be set for the class (the `/Gb` compiler option sets this pragma on at the start of the compilation unit). For further details see "The SOMNoDataDirect Pragma" on page 331.

In DSOM, static data members are local to the current process, they are not managed remotely. DSOM classes must also have a default constructor.

System Object Model (SOM) Options

This section describes the compiler options available for SOM support in VisualAge C++.

SOM options that affect the same settings as SOM pragmas are effective except when overridden by those pragmas. For example, the `/Ga` compiler option, which causes all classes to implicitly derive from SOMObject, turns the **SOMAsDefault** pragma on at the start of the translation unit. This pragma remains in effect until a **#pragma SOMAsDefault(off|pop)** is encountered in the translation unit. See "Conventions Used by the SOM Pragmas" on page 313 for more information on the relationship between SOM pragma settings and SOM options.

In addition to the compiler options, the compiler defines a macro, **__SOM_ENABLED__**, whose value corresponds to the level of SOM support provided by the compiler. If SOM support is not provided for a particular release of the compiler, **__SOM_ENABLED__** is not predefined.

The macro's value is a positive integer constant. For the first SOM-supporting release of VisualAge C++, the level of SOM supported is SOM 2.1, so the macro has the value 210.

/Ga

Syntax:	**Default:**
/Ga[+|-]	/Ga-

This option turns on implicit SOM mode, and also causes the file som.hh to be included. It is equivalent to placing **#pragma SOMAsDefault(on)** at the start of the translation unit.

All classes are implicitly derived from SOMObject until a **#pragma SOMAsDefault(off)** is encountered.

For further details see "The SOMAsDefault Pragma" on page 315.

/Gb

Syntax:	**Default:**
/Gb[+|-]	/Gb-

This option instructs the compiler to disable direct access to attributes. Instead, the get and set methods are used. This is equivalent to specifying **#pragma SOMNoDataDirect(on)** as the first line of the translation unit.

For further details see "The SOMNoDataDirect Pragma" on page 331.

/Gz

Syntax:	**Default:**
`/Gz[+|-]`	`/Gz-`

Use this option to initialize SOM classes at their point of first use during the execution of your program.

By default, all SOM classes statically used in your program are initialized at static initialization time. This makes your program smaller, but may result in the initialization of classes that are not dynamically used.

With any setting of this option, any reference to a static member of a SOM class will cause the class to be initialized.

/Xs

Syntax:	**Default:**
`/Xs<`*directory*`|->`	`/Xs-`

Use this option to exclude files in the specified directories when implicit SOM mode is turned on (when classes are implicitly derived from SOM). The syntax of this option is:

```
          ┌──────────┐
►►──/Xs───▼─directory─┴──────────────────────────────────►◄
```

where *directory* is the name of the directory or directories you want to exclude. Directory names are separated with a semicolon (;).

This option is useful for mixing implicit SOM mode with existing include files that include declarations of classes you do not want to be implicit SOM classes.

SOM Options

/Fr

Syntax:	**Default:**
`/Fr<`*classname*`>`	None

Use this option to have the compiler write the release order of the specified class to standard output. The release order is written in the form of a **`SOMReleaseOrder`** pragma. You can capture the output from this option when developing new SOM classes, and include the pragma in the class definition. The syntax of the option is:

```
►►──/Fr──C++ClassName──────────────────────────────────►◄
```

For further details see “Release Order of SOM Objects” on page 281 and “The SOMReleaseOrder Pragma” on page 333.

/Fs

Syntax:	**Default:**
`/Fs[+|-|`*file*`|` *directory*`]`	`/Fs-`

Use this option to have the compiler generate an IDL file if a file with an `.hh` extension is explicitly specified on the command line. The syntax of the option is:

```
►►──/Fs──┬────────────┬────────────────────────────────►◄
         ├─+──────────┤
         ├─-──────────┤
         ├─filename───┤
         └─directory──┘
```

where:

`/Fs<+>` specifies that an IDL file will be created for every `.hh` file that is specified on the command line and is in the current directory. This is the default.

`/Fs` *`filename.ext`* is like `/Fs +`, but the IDL file that is created will have the specified filename. If you do not specify an `ext`, the extension will be `idl`.

`/Fs` *`directory_name`* is like `/Fs +`, but the IDL file that is created will be put in the directory *`directory_name`* rather than the current directory. *`directory_name`* must end with a backslash "\".

`/Fs-` specifies that no IDL file should be created.

Macro Defined for SOM

VisualAge C++ predefines the **`__SOM_ENABLED__`** macro with a positive integer value, to indicate the level of SOM support provided. Currently the value for **`__SOM_ENABLED__`** is 210, which indicates that the level of SOM support described in this chapter is available. If **`__SOM_ENABLED__`** is not defined or has a zero value, SOM is not supported by the version of the compiler on which the program is being compiled.

Pragmas for Using SOM

This section describes the pragmas available for SOM support on VisualAge C++. See the previous sections for background information on the reasons and uses for the pragmas.

Note: The SOM pragmas are case-insensitive. They are shown here in a mixed-case format to make them easier to read. You can use any combination of upper- and lowercase letters for the pragma names and for the **`on`**, **`off`** and **`pop`** arguments. However, you must still enter C++ tokens such as class, method, and data member names exactly as they are declared in your program.

Conventions Used by the SOM Pragmas

Some of the SOM pragmas use certain conventions to specify the scope to which the pragma applies. This section explains those conventions.

Pragmas Containing on | off | pop

SOM pragmas containing an argument of **`on`**, **`off`**, or **`pop`** implement a stack-modelled approach to setting their option. The arguments do the following:

on Pushes the prior state (on or off) of the pragma onto the pragma's "stack," and turns the setting on.

off Pushes the prior state of the pragma onto the pragma's "stack," and turns the setting off.

pop Restores the most recently saved state from the pragma's "stack."

The following example shows the effect of the **`SOMAsDefault`** pragma with different settings:

```
// ...  SOMAsDefault is off, or ON if program compiled with /Ga

#pragma SOMAsDefault(on)
// ...  SOMAsDefault now on

#pragma SOMAsDefault(pop)
// ...  SOMAsDefault now off, or ON if program compiled with /Ga

#pragma SOMAsDefault(off)
// ...  SOMAsDefault now off

#pragma SOMAsDefault(pop)
// ...  SOMAsDefault now off, or ON if program compiled with /Ga
```

It is recommended that **on** or **off** be used only at the beginning of a block, and **pop** only at the end of the block. This ensures that default settings are preserved around your own settings.

If you **pop** a pragma more times than you push it with **on** or **off**, the results are unpredictable.

Pragmas Containing an Asterisk (*)

Certain SOM pragmas accept either a C++ class name or an asterisk (*) as one of their arguments. You can use the asterisk to indicate that the class the pragma applies to is the class within which the pragma occurs. For example:

```
#pragma SOMAsDefault(on)
class A {
   //...
#pragma SOMClassVersion(*,3,1)
// Version number applies to class A
}

Class B {
   // ...
#pragma SOMClassVersion(B,3,3)
// Could have specified * instead of B
}

#pragma SOMClassVersion(*,2,5)
// Error - not in the scope of any class!
```

The SOM Pragma

This pragma causes the compiler to recognize the `SOMObject` class as the special base for all SOM classes.

Note: The compiler still requires a full declaration for `SOMObject`. Therefore, you must include the header file containing this declaration.

There should only be one occurrence of this pragma, and it should be placed in the same header file in which `SOMObject` is declared.

The syntax of the pragma is:

```
►►──#pragma SOM─────────────────────────────────────────────────►◄
```

The SOMAsDefault Pragma

The setting of this pragma determines how the compiler should treat classes that are not explicitly derived from `SOMObject`. When the pragma is in effect, all non-local classes are implicitly derived from `SOMObject`. When the pragma is not in effect, classes must be explicitly derived from `SOMObject` in order to be supported for use by SOM programs.

The syntax of the pragma is as follows:

```
►►──#pragma SOMAsDefault(──┬─on──┬──)───────────────────────────►◄
                           ├─off─┤
                           └─pop─┘
```

The **`on`** argument saves the current setting, and turns **`SOMAsDefault`** on. The **`off`** argument saves the current setting, and turns **`SOMAsDefault`** off. The **`pop`** setting restores the most recently saved but still unrestored setting. See “Pragmas Containing on | off | pop” on page 313 for more information on how to use these arguments.

When this pragma is turned on for the first time in a compilation unit, it also causes the `som.hh` header file to be included if it has not already been included.

The `/Ga` compiler option provides the same effect as setting **`#pragma SOMAsDefault(on)`** at the start of the translation unit.

The SOMAttribute Pragma

Use this pragma to specify that a data member is an attribute, and to communicate IDL information regarding the implementation of attributes. For an explanation of how attributes are used, see “set and get Methods for Attribute Class Members” on page 290. The syntax of the pragma is:

SOM Pragmas

```
►►──#pragma SOMAttribute(──DataMember──,──────────────────────►

     ┌─,────────────────┐
►────▼─┬─indirect───────┬─┴──)──────────────────────────────────►◄
       ├─nodata─────────┤
       ├─noget──────────┤
       ├─noset──────────┤
       ├─privatedata────┤
       ├─protectedata───┤
       ├─publicdata─────┤
       ├─readonly───────┤
       └─virtualaccessors┘
```

The pragma must appear within the class definition or declaration in which the data member is defined. Each attribute in a class must be defined in its own pragma. You can only make a non-static data member into an attribute. The member cannot be a reference to an abstract class because the `_get`/`_set` functions have to operate on values. The keywords have the following effects:

indirect	The interface (prototype) for the get and set methods of this attribute must use one level of indirection for both the argument to be set and the return from the get. This means that if the type is normally passed and returned by value, it will have its address returned instead. For example, `T _get_X()` actually returns `*T`, and `_set_X(T)` actually accepts `*T` as argument. **`indirect`** is ignored for structs and arrays.
nodata	The compiler does not allocate any instance data corresponding to this attribute, and does not generate definitions for the get and set methods. This means that you must define these methods yourself and allocate any instance data these methods require. **`nodata`** implies that there is no way for C++ code to take the address of this variable. The compiler issues an error message when you attempt to do this.
	You must write and declare the corresponding get and set functions, **`_get_`***`variable`* and **`_set_`***`variable`*, where *`variable`* is the attribute's name.
noget	The compiler does not generate a body for the attribute's get method. You must provide a body for the get method.
noset	The compiler does not generate a body for the attribute's set method. You must provide a body for the set method. This qualifier is ignored if the attribute is const.

privatedata	The compiler defines instance data for the member class and gives it private access. This is the default.
protectedata	The instance data for the member class has protected access.
publicdata	The instance data for the member class has public access.
readonly	The attribute is not allowed to have a set method. The compiler does not generate one. If you provide one, the compiler flags it as an error.
virtualaccessors	The `_get/_set` methods will be virtual functions. By default, `_get` and `_set` are nonvirtual functions.

The access for the `_get/_set` methods is the same as the access for the data member. For example, access for the `_get/_set` methods of a protected data member are protected. By default, access to the data itself is private unless you specify otherwise with one of the `protecteddata` or `publicdata` keywords. If you do not use the **`SOMAttribute`** pragma, the data member is not an attribute. Attribute qualifiers **`nodata`**, **`privatedata`**, **`protectedata`**, and **`publicdata`** are mutually exclusive. It is an error for the access of an attribute's instance data to be greater than the access of the attribute. For example, it is an error for a private attribute to have public instance data.

If you do not use the **`SOMNoDataDirect`** pragma, access to data members uses direct access if the user code has access to the instance data.

When **`SOMNoDataDirect`** is used, the `_get/_set` methods are used. The access for the `_get/_set` methods is the same as the access for the data member. For example, access for a **`protected`** data member's `_get` and `_set` methods would be **`protected`**.

The **`nodata`** attribute modifier and the **`SOMNoDataDirect`** pragma have different effects, although their names are similar.

Normally, the compiler creates instance data in the class to implement an attribute, and generates definitions for get and set methods that access this "backing" data. The access class of the methods is that of the attribute, but the backing data is **`private`**. You can override this with the **`publicdata`** or **`protecteddata`** modifiers.

If you do not specify other modifiers or pragmas, then uses of the attribute are compiled either into direct accesses of the backing data, or into calls to the get and set methods. The compiler determines whether the code using the attribute can "see" the backing data, according to the usual C++ access rules. Because members and friend functions of a class do have access to its private data, they directly access any backing data for attributes of that class. Methods in derived classes only have access to public and protected members of a base class, so can only access backing data that is public or protected. Private backing data in a base class is not accessible, so uses of public or protected attributes with private backing data must call _get and _set.

When you add the **nodata** modifier to an attribute, the compiler no longer automatically creates backing data, and only declares the get and set methods. You must supply definitions for them. Also, uses of the attribute will always be compiled into get or set calls.

When you use the **SOMNoDataDirect** pragma on a class, it does not affect the generation of methods or backing data, but it does affect how uses of the attribute are compiled. **SOMNoDataDirect** is an indication to the compiler that instances of the class may be proxies for remote objects built by DSOM. Because direct data access is not possible for remote objects, the compiler must then generate _get/_set calls for all attribute uses, unless the object is known to be local. The only object that can be safely assumed local is the object pointed to by **this**, so direct data access only happens for accesses through the **this** pointer. This condition is imposed in addition to the access checks described above.

The SOMCallStyle Pragma

Use this pragma to specify the callstyle of the class within which the pragma occurs. The syntax of this pragma is:

```
►►──#pragma SOMCallStyle(──┬─OIDL─┬──)──────────────────────────►◄
                           └─IDL──┘
```

The **OIDL** option indicates that the callstyle of methods introduced by the class does not include the `Environment*` argument, while the **IDL** option indicates that the callstyle does include the `Environment*` argument. The default is for IDL callstyle to be used.

For further details see “IDL and OIDL Callstyles” on page 293.

The SOMClassInit Pragma

Use this pragma to specify a function that the SOM runtime is to invoke during creation of the class object for the named class. The syntax of this pragma is:

```
►►──#pragma SOMClassInit(──┬─*────────────┬──,──C++Prototype───►
                           └─C++ClassName─┘
►──)─────────────────────────────────────────────────────────►◄
```

The asterisk indicates that the pragma applies to the innermost enclosing class within which the pragma is found.

The *C++Prototype* is a C++ function prototype without the return type. For example, the function `double sqrt(double)` would appear as `sqrt(double)` in this pragma.

A class object is created for a class when the first object of that class is created. The function called after the class object is created must have the following form:

```
void FunctionName(SOMClass*);
```

The name of the function is not significant. Once you have declared or defined this function, you can associate it with the class constructor for a class using the pragma:

```
#pragma SOMClassInit(FunctionName)
```

You do not need to use this pragma unless you want to define a function to be called when the class object is created.

The SOMClassName Pragma

Use this pragma to specify SOM names for C++ classes and template classes. You should keep in mind that naming in SOM is not case sensitive, so any names you supply through **SOMClassName** should be distinguishable from other names regardless of case. In addition, the Common Object Request Broker Architecture (CORBA) requires that names begin with a letter of the alphabet.

If you do not use the **SOMClassName** pragma, *and* the **SOMNoMangling** pragma is not in effect for the class, the compiler mangles the class name, which may make the class difficult to use from non-C++ programs. Mangled names tend to be nonobvious, and accessing them from SOM programs can reduce code readability and increase the likelihood of coding errors.

The syntax of the **SOMClassName** pragma is:

```
►►──#pragma SOMClassName(──┬─*────────────┬──,──"──────────────►
                           └─C++ClassName─┘
►──NameOfSomClass──")─────────────────────────────────────────►◄
```

The asterisk indicates that the pragma applies to the innermost enclosing class within which the pragma is found.

For example:

```
#pragma SOMAsDefault(on)
class MyCppClass { /* ... */ };
#pragma SOMClassName(MyCppClass, "MySOMClass")
class AnotherClass {
#pragma SOMClassName(*,"AnotherSOMClass")
//...
};
```

The requirements for the **SOMClassName** pragma are:

- The class in question must already have been declared when the compiler encounters the pragma.
- The class must be a SOM class.
- The SOM class name cannot be the same as a name associated with a different SOM class. This means that you cannot write code such as the following:

  ```
  class x : SOMObject { int a; };
  class y : SOMObject { int b; };
  #pragma SOMClassName(x,"y") // error - there is already a SOM Y class.
  ```

  The compiler will catch this error if the two SOM classes involved are in the same compilation unit. If they are in separate compilation units, the compiler will not issue an error message, and the results of the program are unpredictable.
- The pragma must appear before the compiler needs to access the class to allocate an instance of the class or one of its subclasses.
- If the asterisk (*) is used, the pragma must appear within the declaration for a SOM class.

Multiple equivalent **SOMClassName** pragmas are ignored. The compiler issues an error if it detects multiple **SOMClassName** pragmas for the same class that are not equivalent.

The SOMClassVersion Pragma

SOM supports explicit version numbering for classes. The SOM runtime uses this information to ensure that the classes of a SOM library are at least as recent as the version of the library a client program was compiled to. When you use the **SOMClassVersion** pragma, you prevent the compiler from providing version n of a class when a client program was expecting version n+1. See “Version Control for SOM Libraries and Programs” on page 284 for a more in-depth explanation of class versioning. The syntax of the pragma is:

```
►►──#pragma SOMClassVersion(──┬─C++ClassName─┬──,──Major──,──────►
                              └─*────────────┘

►──Minor──)──────────────────────────────────────────────────────►◄
```

You can use the asterisk (*) to indicate that the pragma applies to the innermost enclosing class within which the pragma occurs. If you use the *C++ClassName* form of the pragma, the class must already have been declared at the point where the pragma is encountered.

In the following example, class Q is given a major version of 3 and a minor version of 2:

```
#pragma SOMAsDefault(on)
class Q {
   public:
   //...
#pragma SOMClassVersion(*,3,2)
};
#pragma SOMAsDefault(pop)
```

The following considerations apply to this pragma:

- Both the major and minor version numbers must be provided, and both must be positive or zero-valued integers.
- The compiler issues an error message if you specify multiple conflicting **SOMClassVersion** pragmas for a given class.
- The class must already be declared at the point where the pragma is encountered.
- In the absence of a **SOMClassVersion** pragma for a class, the compiler assumes zero for both version levels.

The SOM runtime treats a zero version value for a class as indicating that versions do not matter, and consequently does not check for version compatibility.

The SOMDataName Pragma

Use this pragma to specify SOM names for C++ class data members. You only need to use this pragma if you want access to the class of the applicable data member from non-C++ programs. If you do not use this pragma or the **SOMNoMangling** pragma, data member names are mangled by the compiler, and the mangled names can lead to coding errors in the non-C++ programs that attempt to use them (because the names are obscure and typically very long). If the member is an attribute, the member's SOM name is used to form the get and set method names.

The syntax of the pragma is:

```
►►──#pragma SOMDataName(──C++DataMember──,──"SomName"──)──────────►◄
```

This pragma may only occur within the body of the corresponding class declaration, and only after the corresponding data member has been declared.

The SOMDefine Pragma

Use this pragma in classes you define that have all member functions inline. The pragma is not necessary for classes that have at least one non-inline member function. This pragma (or the point at which the compiler encounters the definition for the first out-of-line function declared within the class) causes the compiler to emit the **SOMBuildClass** data structures, which are used by the SOM runtime. The **SOMDefine** pragma for a class with all inline functions can occur in any compilation unit, but should only appear once across all compilation units. The syntax of the pragma is:

```
►►──#pragma SOMDefine(──┬─*────────────┬──)──────────────────►◄
                        ├─on───────────┤
                        ├─off──────────┤
                        ├─pop──────────┤
                        └─C++ClassName─┘
```

You can use the asterisk (*) to indicate that the pragma applies to the innermost enclosing class within which the pragma occurs. This version of the pragma does not apply to nested classes of the class where the pragma occurs.

For the **C++ClassName** version, the name of the specified class must be visible at the point where the pragma is encountered.

The **on**, **off**, and **pop** settings are independent of the asterisk setting. Use them to control the default over specific ranges of source. (See "Pragmas Containing on | off | pop" on page 313 for information on how to use these arguments.)

If a **SOMDefine(*)** pragma occurs within the body of a class, that class will be defined (assuming it has no out-of-line functions) regardless of the current value set by **on/off/pop**.

Classes that have all member functions defined inline are considered declarations by the C++ language rules. This means that such classes can be "declared" in several compilation units. Normally, the compiler would have to create a class structure and its data and method tables each time it encounters such a class. When you use the **SOMDefine** pragma, you allow the compiler to create only one copy of the class structure, which can reduce your program's storage requirements and improve performance.

This pragma is ignored if the class has any out-of-line member functions.

The SOMIDLDecl Pragma

Use the **SOMIDLDecl** pragma to override the IDL declaration that the compiler would otherwise generate for the named type or member function to which the pragma applies. You can use this pragma to include information related to IDL contexts and IDL exceptions, or to fine-tune translations between **char*** and **string** types. The syntax of the pragma is:

```
►►──#pragma SOMIDLDecl(──┬─C++TypeName──┬──"────────────────────►
                         └─C++Prototype─┘
►──"IDLDeclaration"──)─────────────────────────────────────────►◄
```

The type or member function named must be defined before the pragma is encountered. For type names, the sequence %N within the string is replaced by the name of the type.

The *C++Prototype* is a C++ function prototype without the return type. For example, the function double sqrt(double) would appear as sqrt(double) in this pragma. If the prototype has a trailing **const**, you must include this in the prototype.

The following example shows a use of this pragma. Here the pragma redeclares one of the **char*** arguments of method P as type **string**, while keeping the other as type **char***, and indicates that the method may raise exception excl.

```
typedef int type1;
#pragma SOMIDLDecl (type1, "typedef foobar type1");

class T : public SOMObject {
   public:
      void P(char*, char*);
      #pragma SOMMethodName(P,"P")
      #pragma SOMIDLDecl(P, "void P(string, char*) raises(excl)")
      // ...
   };
```

The SOMIDLPass Pragma

Use the **SOMIDLPass** pragma to emit arbitrary text to IDL. The syntax of the pragma is:

```
►►──#pragma SOMIDLPass(──┬─*────────────┬──"──Label──","───────►
                         └─C++ClassName─┘
►──StringToEmit──")────────────────────────────────────────────►◄
```

The *Label* field indicates where in the IDL file for a class the string is to be emitted to. The possible labels are described below. The pragma accepts any combination of upper- and lowercase characters for a label:

`Begin`	Text is emitted at the start of the IDL file, just after the controlling `#ifdef` and `#define` pair of directives.
`End`	Text is emitted at the end of the IDL file, just before the controlling `#endif` directive.
`Implementation-Begin`	Text is emitted right after the opening brace of `implentation {`.
`Implementation-End`	Text is emitted just before the closing brace of the implementation section.
`Interface-Begin`	Text is emitted at the start of the interface section for the class, right after the opening { brace.
`Interface-End`	Text is emitted at the end of the interface section of the class, immediately *before* the implementation section.
Other text	The compiler ignores a **SOMIDLPass** pragma whose label does not match one of the above. No warning is given.

The following class definition shows uses of the **SOMIDLPass** pragma:

```
class T : public SOMObject {
      public: boolean somRespondsTo(somId);
      #pragma SOMIDLTypes(*, boolean, somId)
      #pragma SOMIDLPass(*,"Begin", "//Top")
      #pragma SOMIDLPass(*,"End", "//End")
      #pragma SOMIDLPass(*,"Interface-Begin", "//Int Begin")
      #pragma SOMIDLPass(*,"Interface-End", "//Int End")
      #pragma SOMIDLPass(*,"Implementation-Begin", "//Imp Begin")
      #pragma SOMIDLPass(*,"Implementation-End", "//Imp End")
   };
   #pragma SOMIDLPass(T,"Interface-Begin", "//Int Begin 2")
   #pragma SOMIDLPass(T,"Interface-End", "//Int End 2")
   #pragma SOMIDLPass(T,"Implementation-Begin", "// ** Imp Begin 2")
```

This example causes IDL to be emitted that looks like the following;

```
#ifndef T__IDL__
#define T__IDL__

// Top
#include <som.hh>

typedef int boolean;
typedef void* somId;

interface T : SOMObject {
   // Int Begin
   // Int Begin 2
   void f();
   // Int End
   // Int End 2
   Implementation {
      // Imp Begin
      // ** Imp Begin 2
      somRespondsTo : override;
      // ...
      // Imp End
   };
};

// End
```

SOMIDLPass pragmas are cumulative; each one adds to the text emitted for the class. The relative order of the pragmas are retained.

The SOMIDLTypes Pragma

Use the **SOMIDLTypes** pragma to list types that you want the compiler to emit when it generates IDL for the specified class. The syntax of the pragma is:

```
                                                  ┌─,────────┐
►►──#pragma SOMIDLTypes(──┬─*────────────┬──────▼─typeName─┴──)────►◄
                          └─C++ClassName─┘
```

The asterisk indicates that the pragma applies to the innermost enclosing class within which the pragma is found.

The following directive:

```
#pragma SOMIDLTypes(MyClass, size_t, AnotherType)
```

Would ensure that size_t and AnotherType are emitted whenever IDL is emitted for class MyClass.

Uses of this pragma for a given class are cumulative. This means that each such pragma for a class adds the specified types to the list for the class, and the order in which the types are emitted is the same as the order in which they are encountered. By default (i.e., if no **SOMIDLTypes** pragma is specified), only the class itself is emitted.

Note: When the compiler encounters a **SOMIDLTypes** pragma for a type that is defined in a nested include file, it generates only an **#include** of the IDL file corresponding to that nested include file. In other words, classes and typedefs defined in nested include files are not generated directly in the IDL file. Note that the IDL for the nested include file must be generated separately, because the compiler only generates IDL for declarations in the file being compiled.

The SOMMetaClass Pragma

Use this pragma if you want to identify a particular class for SOM to use as the metaclass of a SOM-enabled C++ class. For more information on SOM metaclasses, see “Metaclasses” on page 300. The syntax of the pragma is:

```
►►──#pragma SOMMetaClass(──┬─C++ClassName─┬──,──────────────────►
                           └─*────────────┘

►──┬─*──────────────────┬──)──────────────────────────────────►◄
   ├─"SOMClassName"─────┤
   └─C++MetaClassName───┘
```

The *C++ClassName* indicates what class is to have the specified metaclass as its metaclass. This form of the pragma can occur at any scope. The names of all specified C++ classes must be visible.

An asterisk (*) in the first position indicates that the innermost enclosing class within which the pragma occurs is the class that will have the specified metaclass. An asterisk in the second position indicates that the innermost enclosing class within which the pragma occurs is the class that will be the metaclass for the specified class. You should never use the asterisk in *both* positions at once; this may cause the program to enter an infinite loop when an object of the class is created. In the following example, class Mountain is given a metaclass of Rock, and class Tree is given a metaclass of Plant:

```
class Mountain: public SOMObject { // ...
   #pragma SOMMetaClass(*,Rock)
}
class Plant: public SOMObject { // ...
   #pragma SOMMetaClass(Tree,*)
}
class Loop: public SOMObject { // ...
   #pragma SOMMetaClass(*,*) // Error - will loop infinitely
}
```

In the version of the pragma that takes a SOM class name as the metaclass, the SOM class name must be enclosed in double quotation marks. In the version that takes a C++ class name as the metaclass, the metaclass must not be enclosed in double quotation marks.

In the absence of a **SOMMetaClass** pragma, the compiler operates as if `SOMClass` was specified as the metaclass.

The compiler issues an error message if you use multiple inequivalent **SOMMetaClass** pragmas for a class.

The SOMMethodAppend

Use the **SOMMethodAppend** pragma to generate the IDL “context” and “raises” strings for methods. The syntax of the pragma is:

```
►►──#pragma SOMMethodAppend(──────────────────────────────────────►
►──C++FunctionPrototypeLessReturn,"string"──)────────────────────►◄
```

The contents of the string will be collected and emitted at the end of the IDL for the function. The “context” information will be used for CORBA contexts and exceptions. If the pragma is used more than once for a given method, the strings will be concatenated.

The following example illustrates how this pragma is used. Given:

```
class X : public SOMObject {
    void MyNewMethod(int, float);
    #pragma SOMNoMangling(*)
    #pragma SOMMethodAppend(MyNewMethod, "raises(\"this, that\")")
    #pragma SOMMethodAppend(MyNewMethod, "context(\"something\")")
};
```

the following fragment of IDL will be produced:

```
void MyNewMethod(in long p__arg1, in float p__arg2)
  raises("this","that") context("something");
```

The SOMMethodName Pragma

Use this pragma to specify SOM names for C++ methods and operators. You only need to use this pragma if you want access to the class of the applicable method from non-C++ programs. If you do not use this pragma or the **SOMNoMangling** pragma, method names are mangled by the compiler, and the mangled names can lead to coding errors in the non-C++ programs that attempt to use them (because the names are obscure and typically very long).

The syntax of the pragma is:

```
►►──#pragma SOMMethodName(──┬─C++Prototype───┬──,──────────────►
                            └─C++FunctionName─┘
►──"SomMethodName"──)─────────────────────────────────────────►◄
```

The *C++Prototype* is a C++ function prototype without the return type. For example, the function `double sqrt(double)` would appear as `sqrt(double)` in this pragma. If the prototype has a trailing **const**, you must include this in the prototype.

The *C++FunctionName* is an unambiguous C++ function name (one that is not overloaded within the class). You do not include the function's signature. If you use this version of the pragma for a function that has more than one overloaded version in a class, the compiler issues an error message.

If you do not need to access the class from non-C++ programs, you do not need to use either **SOMMethodName** or **SOMNoMangling** for the class.

Note: These pragmas change the SOM name of a method. As discused in "SOM and Upward Binary Compatibility of Libraries" on page 280, renaming an item is equivalent to removing it and adding a new item with the same characteristics. If there is a possibility that you will access the class from non-C++ programs, use the **SOMMethodName** or **SOMNoMangling** pragmas in your initial implementation.

You can use a combination of **SOMMethodName** and **SOMNoMangling** to give unmangled names to methods of a class that non-C++ programs will access. The **SOMNoMangling** pragma (see "The SOMNoMangling Pragma" on page 332) specifies that the C++ name of a method becomes the SOM name of that method. As long as the method is not an overloaded method or an operator other than the default assignment operator, **SOMNoMangling** makes the method accessible to non-C++ programs by its C++ name after folding all letters to lowercase. The following example shows a class declaration with a combination of **SOMNoMangling** and **SOMMethodName** pragmas:

```
   #pragma SOMAsDefault(on)
   class Address {
      public:
         char* Street;
         int   Phone;
   #pragma SOMNoMangling(on)
         int   call();  // remains as call
         void  print(); // remains as print
   #pragma SOMNoMangling(pop)
         void  update(char* street);
   #pragma SOMMethodName(update(char),"updatestreet")
                     // becomes updatestreet
         void  update(int phone);
   #pragma SOMMethodName(update(int),"updatephone")
                     // becomes updatephone
   };
#pragma SOMAsDefault(pop)
```

The example uses **SOMNoMangling** to cause the C++ methods `call` and `print` to be given SOM names identical to their C++ method names. The example then explicitly renames the different overloads of `update` using **SOMMethodName**, so that calls to those methods from non-C++ programs can be resolved.

You should keep in mind that naming in SOM is not case sensitive, so any names you supply through **SOMMethodName** should be distinguishable from other names regardless of case. In addition, the Common Object Request Broker Architecture (CORBA) requires that names begin with an alphabetic character. If you use the **SOMMethodName** pragma on a method, make sure the SOM name starts with an alphabetic character.

The requirements for the **SOMMethodName** pragma are:

- The pragma must occur in the compilation unit that defines the class (the compilation unit that contains a **SOMDefine** pragma or the first noninline function for the class).
- The method must already have been declared at the point where the pragma is encountered.
- The class must be a SOM class.
- You cannot rename two method signatures in a class to the same name. The compiler issues an error if you attempt this.

- The name of the member function within the **SOMMethodName** pragma must be fully qualified if the pragma occurs outside of the class declaration. For example, function clear() of class Buffer must be specified as Buffer::clear().
- A method may only be renamed in conjunction with the class that introduces it. This means that you cannot rename a function func() in subclass B of class A, if func() was introduced by A.
- You cannot rename a method to _get_X() or _set_X(), where X is the name of an attribute for that class. For example, you cannot do the following:

```
class MyClass : SOMObject {
   public:
      int i;
      int foo();
#pragma SOMAttribute(i)
#pragma SOMMethodName(foo(),"_get_i") // error
   };
```

  because the **SOMAttribute** pragma predefines a get and set method for i. If i were a member of a base class of MyClass rather than of MyClass itself, the above **SOMMethodName** pragma would work, but the compiler would resolve all calls to _get_i() by calling the get method of the base class, rather than by calling foo().

The compiler generates an error message if more than one version of an overloaded SOM function is found and no SOMMethodName pragma has been used to rename versions of the function. The error occurs whenever the compiler detects a version of the function with a signature different from that of the first instantiated version. The error refers to name clashes. You can avoid this error by using **SOMMethodName** before any overload of a function other than the first is used.

Note that different instantiations of templates used as SOM classes may have different names for a method, if **SOMMethodName** is used on the method for a given instantiation of the template. For example:

```
template class A<T> : public SOMObject {
   public:
      Print();
};
#pragma SOMMethodName(A<int>::Print,"PrintInt")
#pragma SOMMethodName(A<char*>::Print,"PrintString")
```

SOMMethodName and Inheritance

If you rename a method of a class using the **SOMMethodName** pragma, a method of a derived class, with the same method signature, has the same SOM method name as specified by the pragma.

The SOMNoDataDirect Pragma

Use this pragma to have the compiler use get/set methods for instance data access. See “set and get Methods for Attribute Class Members” on page 290 for further details.

The syntax of the pragma is:

```
►►──#pragma SOMNoDataDirect(──┬─*───┬──)──────────────────────►◄
                              ├─on──┤
                              ├─off─┤
                              └─pop─┘
```

When this pragma is in effect, all public data members can be accessed by get and set methods only, except as specified below. When the pragma is not in effect, nonprivate data members can be accessed directly, or by the get and set methods. However, if a data member has **#pragma SOMAttribute(nodata)** set, the data member can only be accessed by the get and set methods.

Direct access may be used by the following functions, regardless of the setting of this pragma:

- Methods of the class (methods can access their own instance data directly through the **this** pointer)
- Methods of subclasses, again through the **this** pointer.

Friend classes and methods may use direct access if the pragma is explicitly turned on within the class declaration (using **#pragma SOMNoDataDirect(*)**). If the pragma is turned on implicitly (using **#pragma SOMNoDataDirect(on)**), friend classes and methods must use the get and set methods.

The asterisk (*) indicates that the pragma applies to the innermost enclosing class within which the pragma occurs. The asterisk version of the pragma temporarily overrides any setting obtained by using the **on**, **off**, or **pop** arguments for the pragma, but only for the class in which it occurs. It has no effect on nested classes.

The **on**, **off**, and **pop** arguments are not allowed within the scope of a class. See “Pragmas Containing on | off | pop” on page 313 for more information on how these arguments are used.

The /Gb compiler option is equivalent to specifying **#pragma SOMNoDataDirect(on)** at the beginning of the compilation unit.

If this pragma is in effect when an instance of a SOM class is used by client code, all SOM object data accesses via pointer or reference (other than those that use the **this** pointer) are done indirectly. SOM object data member accesses done through local or global SOM objects may be done directly.

The SOMNoMangling Pragma

Use this pragma to tell the compiler not to mangle the C++ names of methods, static member functions, or instance data when creating SOM names or generating IDL. The syntax of the pragma is:

```
►►──#pragma SOMNoMangling(──┬─*───┬──)──────────────────────►◄
                            ├─on──┤
                            ├─off─┤
                            └─pop─┘
```

See "Conventions Used by the SOM Pragmas" on page 313 for information on how to use the pragma's arguments. Note that, when the asterisk (*) is used in the pragma, settings of the pragma via **on**, **off**, or **pop** are ignored, but only for the class in which the pragma appears with the asterisk. This applies even if **on**, **off**, or **pop** are used within the class itself. However, the asterisk version does not affect nested classes.

When the pragma is in effect, the compiler does the following:

- Generates lowercase versions of declared method names, with no mangling applied. This means that method names do not identify their arguments and class.
- Detects clashes of generated names within a class. This means that two overloaded versions of method `f`, for example `f(int)` and `f(double)`, result in a compiler error message. To correct such a situation, you can use the **SOMMethodName** pragma on all but one of the conflicting methods.

Notes:

1. The pragma does not apply to compiler-generated functions, which continue to use mangled names.
2. User-written member functions that begin with an underscore (except `_get` and `_set` members) are always mangled.
3. It is an error to remap two different C++ signatures to the same SOM name. This can happen, for example, in a class with overloaded methods where **SOMNoMangling** is in effect. In such cases, you should use a **SOMMethodName** pragma to rename all but one of the overloaded methods. A **SOMMethodName** pragma always takes precedence over a **SOMNoMangling** pragma.

The pragma only applies to methods introduced by a class, not to inherited methods. If **SOMNoMangling** is in effect when the compiler encounters a base class, the methods of the base class will have unmangled names, as will methods with the same signatures in any derived class, regardless of the state of **SOMNoMangling** in the derived class.

In the following example, MyNewMethod receives a SOM name of mynewmethod, rather than the mangled version VisualAge C++ would normally generate:

```
#pragma SOMNoMangling(off)
// ...
class X : public SOMObject {
#pragma SOMNoMangling(*) // overrides SOMNoMangling(off)
                         // for entire class
    // ...
    void MyNewMethod(int, float);
};
```

The SOMNonDTS Pragma

Warning:

This pragma is not intended to be used by programmers. Do not use this pragma in your programs, or the results will be unpredictable.

This pragma is automatically inserted in generated .hh files to inform the compiler that the class it applies to was originally a SOM class, and not a C++ class converted to a SOM class by the VisualAge C++.

The SOMReleaseOrder Pragma

Use the **SOMReleaseOrder** pragma to make your SOM classes upward binary compatible (so that client programs can use newer versions of your library without having to recompile their source code each time you issue a new version of the library). When you extend a class, you can only achieve binary compatibility for users of the class if any added functions or data members are placed at the end of the release order list specified in the pragma. See “Release Order of SOM Objects” on page 281 if you want a better understanding of how release order is used to ensure upward binary compatibility.

The syntax of the pragma is:

SOM Pragmas

```
►►──pragma SOMReleaseOrder(──────────────────────────────────────►

    ┌─,──────────────────────────────────────┐
►───▼─*──────────────────────────────────────┴──)────────────►◄
      ├─StaticDataMember────────────────────┤
      └─┬───────┬──┬─Attribute──────────────┤
        └── ! ──┘  ├─C++MemberFunctionPrototype─┤
                   ├─C++UnambiguousFunctionName─┤
                   └─"SOMMethodName"────────────┘
```

The pragma must appear within the body of the class declaration. It contains a comma-separated list of release order elements. A release order element may be any of the following:

- *An asterisk* (*). The asterisk reserves a slot in the release order so that you can later add a member function or data member at that position in the list, without requiring client programs to be recompiled. You can also reserve slots for things like private members that you do not want to expose to client code.
- *An attribute* This uses two slots in the release order, one for the attribute's get method, and one for its set method. Both slots are used even for const data members, which do not have a set method, so that you can later change the method to non-const without breaking binary compatibility. Regardless of whether you define get and set methods or let the compiler generate them for you, you can place either the data member name, or the get and set method names, in the release order. (You cannot specify *both* the data member name and the set and get methods.) For new classes, you should use the data member name, for the sake of code readability and to ensure that the get and set methods for an attribute are always consecutive in the release order. For older SOM classes where you did not allocate consecutive slots for the get and set methods in the class's release order, you must continue to specify each method separately in the correct order.
- *A static data member name*. This uses one slot, for a pointer to the static data member.
- A C++ member function prototype, excluding the return type. This uses one slot, for a pointer to the function. See below for information on the use of the exclamation point (!). Note that if the function is not overloaded within the class you can use the unambiguous function name (see below).
- An unambiguous function name (one that is not overloaded by the class in question or any of its bases).
- *A SOM method name*, enclosed in quotation marks. This is equivalent to specifying the C++ member function name, except that you must specify the

simple SOM method name without specifying argument types. See below for information on the use of the exclamation point.

Elements Preceded by !

Release order elements preceded by an exclamation point (!) let you assert that a member function is to have a slot reserved for it even if the member function was inherited from a base class. The "!" helps the compiler diagnose unexpected base class evolutions. This can occur when a base class later introduces a virtual method whose signature matches one that is currently introduced by this class. If the method is found in the class's release order without the "!", the compiler issues an error message. If you precede the method with "!", you are asserting to the compiler that you are aware of the method's having moved upward in the inheritance structure. VisualAge C++ preserves binary compatibility in such situations, if you use the "!".

The following examples show two versions of a class hierarchy. In the first version, method `aMethod()` is a member of class `Derived`:

```
class Base : public SOMObject {
    };

class Derived : public Base {
    public:
       void aMethod();
    #pragma SOMReleaseOrder(aMethod())
    };
```

This version compiles successfully, because `aMethod()` is found in the release order of the class that introduced it. Later, a version of `aMethod()` is added to `Base`:

```
class Base : public SOMObject {
   public:
      virtual void aMethod();
    };

class Derived : public Base {
    public:
       void aMethod();
    #pragma SOMReleaseOrder(aMethod())
    };
```

A compilation error occurs for this version, because the release order for class `Derived` contains a method that is no longer introduced by the class (it is now introduced by `Base`). The compiler considers this an error because the **SOMReleaseOrder** pragma does not make the inheritance of `aMethod()` from class `Base` explicit. To solve this problem, change the release order pragma to:

```
#pragma SOMReleaseOrder(!aMethod())
```

This informs the compiler that the programmer coding class `Derived` is aware of the addition of `aMethod()` to class `Base`. The program then compiles successfully.

Other Requirements

This pragma may only appear within the body of the corresponding class definition. Only one such pragma is allowed per class. If you do not provide a release order, the compiler will assume a release order matching the order of declaration within the class body. Although you can avoid having to specify a release order by always placing new methods and data members below existing ones in the private and protected/public sections of the class definition, use of the **SOMReleaseOrder** pragma is strongly recommended for safety and code readability.

Items in the release order list must have been declared prior to the pragma, and must appear only once in the list.

If a **SOMReleaseOrder** pragma is given for a class, it must list all the methods and data members introduced by that class. (Compiler-generated methods, such as the four default assignment operators that the compiler provides if you do not define any, must also be listed, if you want to take their address.) The compiler issues a warning message when it encounters a partial list.

You can use the `/Fr` option to have the compiler generate a **#pragma SOMReleaseOrder** for a class. The release order includes compiler-defined methods. By default the compiler places methods it generates at the end of the release order. For further details see "The SOMReleaseOrder Pragma" on page 333.

Templates and Release Orders

Because the **SOMReleaseOrder** pragma must occur within the declaration for a class, you cannot declare different release orders for different instantiations of a template class. If you rename methods of a template instantiation using **SOMMethodName**, you must still indicate the original C++ name of each method in the release order within the template class. If you want to provide two different release orders for different instantiations of a template, you must make one of the classes a subclass of the template. You can then declare a different release order for that class, using the "!" to indicate your awareness that member functions are derived from a base class.

Part 5. Appendixes

Appendixes

ANSI Notes on Implementation-Defined Behavior

VisualAge C++ product supports the requirements of the *American National Standard for Information Systems / International Standards Organization – Programming Language C* standard, ANSI/ISO 9899-1990[1992], and the current draft of the *Working Paper for Draft Proposed American National Standard for Information Systems - Programming Language C++* ANSI X3J16/92-0060, (September 17, 1992), as understood and interpreted by IBM as of March 1993. It also supports the IBM SAA C standards as documented in the *Language Reference*. This appendix describes how VisualAge C++ behaves where the ANSI C Standard describes behavior as implementation-defined. These behaviors can affect your writing of portable code.

Implementation-Defined Behavior Common to Both C and C++

The following sections describe how the VisualAge C++ product defines the behavior classified as implementation-defined in the ANSI C Standard.

Identifiers

- The number of significant characters in an identifier with no external linkage is 255.
- The number of significant characters in an identifier with external linkage is 255.
- VisualAge C++ compiler truncates all external names to 255 characters.
- Case sensitivity: the case of identifiers is respected unless you link using the `/IGNORECASE` option of ILINK.

Characters

- A character is represented by 8 bits, as defined by the `CHAR_BIT` macro in **<limits.h>**.
- The same code page is used for the source and execution set. (Source characters and strings do not need to be mapped to the execution character set.)
- When an integer character constant contains a character or escape sequence that is not represented in the basic execution character set, the `char` is assigned the character after the backslash, and a warning is issued. For example, `'\q'` is interpreted as the character `'q'`.
- When a wide character constant contains a character or escape sequence that is not represented in the extended execution character set, the `wchar_t` is assigned the character after the backslash, and a warning is issued.

ANSI Notes

- When an integer character constant contains more than one character, the last 4 bytes represent the character constant.
- When a wide character constant contains more than one multibyte character, the last `wchar_t` value represents the character constant.
- The default behavior for `char` is `unsigned`.
- Any sequential spaces in your source program are interpreted as one space.
- All spaces are retained for the listing file.

Strings

- VisualAge C++ compiler provides the following additional sequence forms for `strtod, strtol,` and `strtoul` functions in locales other than the C locale:

  inf   infinity   nan

  All of these sequences are not case sensitive.
- When you use DBCS (with the `/Sn` compiler option), a hexadecimal character that is a valid first byte of a double-byte character is treated as a double-byte character inside a string. A `0` is appended to the character that ends the string. Double-byte characters in strings **must** appear in pairs.

Integers

Figure 30. Integer Storage and Range

Type	Amount of Storage	Range (in <limits.h>)
`signed short`	2 bytes	`-32768` to `32767`
`unsigned short`	2 bytes	`0` to `65535`
`signed int`	4 bytes	`-2147483648` to `2147483647`
`unsigned int`	4 bytes	`0` to `4294967295`
`signed long`	4 bytes	`-2147483648` to `2147483647`
`unsigned long`	4 bytes	`0` to `4294967295`

Note: Do not use the values in this table as numbers in a source program. Use the macros defined in **<limits.h>** to represent these values.

- When you convert an integer to a `signed char`, the least-significant byte of the integer represents the `char`.
- When you convert an integer to a `short` signed integer, the least-significant 2 bytes of the integer represents the `short int.`
- When you convert an unsigned integer to a signed integer of equal length, if the value cannot be represented, the magnitude is preserved and the sign is not.

- When bitwise operations (OR, AND, XOR) are performed on a `signed int`, the representation is treated as a bit pattern.
- The remainder of integer division is negative if exactly one operand is negative.
- When either operand of the divide operator is negative, the result is truncated to the integer value and the sign will be negative.
- The result of a bitwise right shift of a negative signed integral type is sign extended.
- The result of a bitwise right shift of a non-negative signed integral type or an unsigned integral type is the same as the type of the left operand.

Floating-Point Values

Figure 31. Floating Point

Type	Amount of Storage	Range of Exponents (base 10) (in <float.h>)
`float` (IEEE 32-bit)	4 bytes	`-37` to `38`
`double` (IEEE 64-bit)	8 bytes	`-307` to `308`
`long double` (IEEE 80-bit)	16 bytes	`-4931` to `4932`

- When an integral number is converted to a floating-point number that cannot exactly represent the original value, it is truncated to the nearest representable value.
- When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest representable value.

Arrays and Pointers

- The type of the integer required to hold the maximum size of an array (the type of the `sizeof` operator, `size_t`) is `unsigned int`.
- The type of the integer required to hold the difference between two pointers to elements of the same array (`ptrdiff_t`) is `int`.
- When you cast a pointer to an integer or an integer to a pointer, the bit patterns are preserved.

ANSI Notes

Registers

- The VisualAge C++ compiler optimizes register use and does not respect the `register` storage class specifier.
- In C programs, you cannot take the address of an object with a `register` storage class. This restriction does not apply to C++ programs.

Structures, Unions, Enumerations, Bit-Fields

- If a member of a union object is accessed using a member of a different type, the result is undefined.
- If a structure is not packed, padding is added to align the structure members on their natural boundaries and to end the structure on its natural boundary. The alignment of the structure or union is that of its strictest member. If the length of the structure is greater than a doubleword, the structure is doubleword-aligned. The alignment of the individual members is not changed. Packed structures are not padded. See Appendix E, "Mapping" on page 393 for more information.
- The default type of an integer bit field is `unsigned int`.
- Bit fields are allocated from low memory to high memory, and the bytes are reversed. For more information, see Appendix E, "Mapping" on page 393.
- Bit fields can cross storage unit boundaries.
- The maximum bit field length is 32 bits. If a series of bit fields does not add up to the size of an `int`, padding may take place.
- A bit field cannot have type `long double`.
- The expression that defines the value of an enumeration constant cannot have type `long double`.
- An enumeration can have the type `char`, `short`, or `long` and be either `signed` or `unsigned`, depending on its smallest and largest values.

  In C++, enumerations are a distinct type, and although they may be the same size as a data type such as `char`, they are not considered to be of that type.

Qualifiers

- All access to an object that has a type that is qualified as `volatile` is retained.

Declarators

- There is no VisualAge C++ limit for the maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type. The only constraint is your system resources.

Statements

- Because the `case` values must be integers and cannot be duplicated, the maximum number of `case` values in a `switch` statement is 4 294 967 296.

Preprocessor Directives

- The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the character constant in the execution character set.
- Such a constant can have a negative value.
- For the method of searching system include source files (specified within angle brackets) see the *User's Guide*.
- User include files can be specified in double quotation marks, for example `"myheader.h"`. For the method of searching user include files, see the *User's Guide*.
- For the mapping between the name specified in the include directive and the external source file name, see the *User's Guide*.
- For the behavior of each **#pragma** directive, see the online or hardcopy *Language Reference*.
- The `__DATE__` and `__TIME__` macros are always defined as the system date and time.

Library Functions

- In extended mode (the default) and for all C++ programs, the `NULL` macro is defined to be `0`. For all other language levels, `NULL` is defined to be: `((void *)0)`.
- When `assert` is executed, if the expression is false, the diagnostic message written by the `assert` macro has the format:

```
Assertion failed:  expression, file file_name, line line_number
```

ANSI Notes

- To create a table of the characters set up by the `CTYPE` functions, use the program in Figure 32 on page 345. The columns are organized by function as follows:

(Column 1)	The hexadecimal value of the character
AN	`isalnum`
A	`isalpha`
C	`iscntrl`
D	`isdigit`
G	`isgraph`
L	`islower`
(Column 8)	`isprint`
PU	`ispunct`
S	`isspace`
PR	`isprint`
U	`isupper`
X	`isxdigit`

- The value returned by all math functions after a domain error (EDOM) is a NaN.
- The value `errno` is set to on underflow range errors is ERANGE.
- If you call the `fmod` function with `0` as the second argument, `fmod` returns `0` and a domain error.

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

int main(void)
{
   int ch;

   for (ch = 0; ch <= 0xff; ch++)
      {
      printf("%#04X ", ch);
      printf("%3s ", isalnum(ch)  ? "AN" : " ");
      printf("%2s ", isalpha(ch)  ? "A"  : " ");
      printf("%2s",  iscntrl(ch)  ? "C"  : " ");
      printf("%2s",  isdigit(ch)  ? "D"  : " ");
      printf("%2s",  isgraph(ch)  ? "G"  : " ");
      printf("%2s",  islower(ch)  ? "L"  : " ");
      printf("%c",  isprint(ch)  ? ch   : ' ');
      printf("%3s",  ispunct(ch)  ? "PU" : " ");
      printf("%2s",  isspace(ch)  ? "S"  : " ");
      printf("%3s",  isprint(ch)  ? "PR" : " ");
      printf("%2s",  isupper(ch)  ? "U"  : " ");
      printf("%2s",  isxdigit(ch) ? "X"  : " ");

      putchar('\n');
      }
      return 0;
}
```

Figure 32. C Program to Print out CTYPE Characters

ANSI Notes

Error Handling

- See the online *Language Reference* for a list of the runtime messages generated for `perror` and `strerror`. Note that the value of `errno` is not generated with the message.
- See the online *Language Reference* for the lists of the messages provided with VisualAge C++ compiler.
- Messages are classified as shown by the following table:

Type of Message	Return Code
Information	0
Warning	0
Error	12
Severe error	16 or 20 or 99

- Use the `/Wn` compile-time option to control the level of messages generated. There is also a `/Wgrp` compiler option that provides programming-style diagnostics to aid you in determining possible programming errors. See the *User's Guide* for further information on this compiler option.

Signals

- The set of signals for the `signal` function and the parameters and usage of each signal are described in Chapter 14, “Signal and OS/2 Exception Handling” on page 217 and in the *C Library Reference* under `signal`.
- `SIG_DFL` is the default signal, and the default action taken is termination.
- If the equivalent of `signal(sig, SIG_DFL);` is not executed at the beginning of signal handler, no signal blocking is performed.
- Whenever you leave a signal handler, it is reset to `SIG_DFL`.

Translation Limits

The VisualAge C++ compiler can translate and compile programs with the following limits:

Figure 33. Translation Limits

Nesting levels of:	
• Compound statements	• No limit
• Iteration control	• No limit
• Selection control	• No limit
• Conditional inclusion	• No limit
• Parenthesized declarators	• No limit
• Parenthesized expression	• No limit
Number of pointer, array and function declarators modifying an arithmetic, a structure, a union, and incomplete type declaration	• No limit
Significant initial characters in:	
• Internal identifiers	• 255
• Macro names	• No limit
• External identifiers	• 255
Number of:	
• External identifiers in a translation unit	• 1024
• Identifiers with block scope in one block	• No limit
• Macro identifiers simultaneously declared in a translation unit	• No limit
• Parameters in one function definition	• 255
• Arguments in a function call	• 255
• Parameters in a macro definition	• No limit
• Parameters in a macro invocation	• No limit
• Characters in a logical source line	• No limit
• Characters in a string literal	• No limit
• Size of an object (in bytes)	• `LONG_MAX`
• Nested #include files	• 127(C),255(C++)
• Levels in nested structure or union	• No limit
• Enumeration constants in an enumeration	• 4 294 967 296 distinct values

ANSI Notes

Streams and Files

- The last line of a text stream does not require a terminating new-line character.
- Space characters that are written out to a text stream immediately before a new-line character appear when read.
- If Ctrl-Z is found and all remaining source characters to end-of-file are whitespace, Ctrl-Z is silently ignored. Subsequent Cntrl-Z's are considered to be whitespace in this case.

  If Ctrl-Z is found ina string or Lstring, it is taken to be part of the string.

  Any other Ctrl-Z is an illegal charater. An error message is printed and the character is ignored.
- There is no limit to the number of null characters that can be appended to the end of a binary stream.
- The file position indicator of an append mode stream is positioned at the end of the file.
- When a file is opened in write mode, the file is truncated. If the file does not exist, it is created.
- A file of zero length does exist.
- For the rules for composing a valid file name, refer to the documentation for the OS/2 operating system.
- For reading, the same file can be simultaneously opened multiple times; for writing or appending, the file can be opened only once.
- When the `remove` function is used on an open file, `remove` fails.
- When you use the `rename` function to rename a file to a name that exists prior to the function call, `rename` fails.
- Temporary files may not be removed if the program terminates abnormally.
- When the `tmpnam` function is called more than `TMP_MAX` times, `tmpnam` fails and returns `NULL`, and sets `errno` to ENOGEN.
- The output of `%p` conversion in the `fprintf` function is equivalent to `%x`.
- The input of `%p` conversion in the `fscanf` function is the same as is expected for `%x`.
- A `'-'` character that is neither the first not the last character in the `fscanf` scan list (`%[`*`characters`*`]`) is considered to be part of the scan list.

- The possible values of errno on failure of fgetpos are EERRSET, ENOSEEK, and EBADPOS.
- The possible values of errno on failure of ftell are EERRSET, ENOSEEK, EBADPOS, and ENULLFCB.

Memory Management

- If the size requested is 0, the calloc, malloc, and realloc functions all return a NULL pointer. In the case of realloc, the pointer passed to the function is also freed.

Environment

- You can pass arguments to main through argv, argc, and envp.
- If a standard stream is redirected to a file, the stream is fully buffered, with the exception of stderr, which is line buffered. If the standard stream is attached to a character device, it is line buffered.
- When the abort function is called, all open files are closed by the operating system. The buffers are not flushed. Any memory files belonging to the process are discarded.
- When the abort function is called, the return code of 3 is returned to the host environment.
- When a program ends successfully and calls the exit function with the argument 0 or EXIT_SUCCESS, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.
- When a program ends unsuccessfully and calls the exit function with the argument EXIT_FAILURE, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.
- If the argument passed to the exit function is other than 0, EXIT_FAILURE or EXIT_SUCCESS, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.
- For the set of environmental names, see Chapter 1, “Setting Runtime Environment Variables” on page 3 and the *User's Guide*.
- For the method of altering the environment list obtained by a call to the getenv function, see the putenv function in the *C Library Reference*.
- For the format and mode of execution of a string on a call to the system function, see the *C Library Reference* under system.

Localization

- A call to `setlocale(LC_ALL, "")` sets the environment to the C default locale.

Time

- The local time zone and daylight saving time zone are `EST` and `EDT`. See Chapter 1, "Setting Runtime Environment Variables" on page 3 and the `tzset` function in the *C Library Reference* for more information on specifying the time zone.
- The era for the `clock` function starts when the program is started by either a call from the operating system or a call to `system`.

C++-Specific Implementation-Defined Behavior

The following sections describe how the VisualAge C++ product defines the behavior classified as implementation-defined in the ANSI C++ Working Paper.

Classes, Structures, Unions, Enumerations, Bit Fields

- Class members are allocated in the order declared; access specifiers have no effect on the order of allocation.
- Padding is added to align class members on their natural boundaries and to end the class on its natural boundary.
- An `int` bit field behaves as an `unsigned int` for function overloading.

Linkage Specifications

- The valid values for the string literal in a linkage specification are:

`"C++"`	Default
`"C"`	C language linkage

Member Access Control

- Class members are allocated in the order declared; access specifiers have no effect on the order of allocation.

Special Member Functions

- Temporary objects are generated under the following circumstances:
  - During reference initialization
  - During evaluation of expressions
  - In type conversions
  - Argument passing
  - Function returns
  - In `throw` expressions.
- Temporary objects exist until there is a break in the flow of control of the program. They are destroyed on exiting the scope in which the temporary object was created.

Migrating Headers from 16-bit C to 32-bit C/C++.

The following section describes the changes you need to make to existing 16-bit C headers for them to work with both 32-bit C and C++ code.

Keywords

- When migrating headers, rename any C++ keywords appearing in your C code.

  `Keywords` are identifiers reserved by a programming language for special use. In C and C++, `keywords` are case sensitive. That means `asm` may be a reserved word but `ASM` need not be. In addition to language specific keywords, VisualAge C++ also has reserved keywords.

  For further information on C, C++ and VisualAge C++ keywords, see the *Language Reference*.

Structures

- Add #**pragma** `pack` statements around declarations of structures that will be passed to or from 16-bit code. Do not use the `_Packed` keyword because it is not supported by C++.
- Declare integers in the structures as short or `long`, not `int`, so that the structures have the same size and layout in both 16-bit and 32-bit code.
- Create `typedefs` for your structures, and use #**pragma** `seg16` on those `typedefs` to specify that those structures should not cross a 64K boundary when laid out in memory.
- Any structure members that are pointers must be qualified with the `_Seg16` type qualifier. For example, `far *` would be translated to `* _Seg16`. This may even need to be done recursively if the 16-bit code will be manipulating the object pointed at.

ANSI Notes

Function Prototypes

- Prototype your functions using the linkage convention keywords. Do not use **#pragma** `linkage` because it is not supported in C++.
- A function pointer that is passed as an argument to a function. When you use a function pointer as an argument to a second function, specify the linkage of the function pointed at in the second function's prototype. This avoids errors when the `/Mp` or `/Ms` compiler options are used to set the default linkage.
- If you pass a pointer to a 16-bit function as a member of an aggregate, array, or class, you must qualify the pointer with **_Seg16**. The **_Seg16** keyword is also required if you use two or more levels of indirection (for example, a pointer to a pointer). If you pass the pointer directly as a parameter, the compiler automatically converts it to a 16-bit pointer and the **_Seg16** keyword is not required.

Required Conditional Compilation Directives

The following directives must be added to the beginning of each header file:

```
#if __cplusplus
extern "C" {
#endif
```

The following directives must be added to the end of each header file:

```
#if __cplusplus
}
#endif
```

Migrating Headers from 32-bit C Set/2 V1.0 to 32-bit C++

You need to make the following changes to your existing header files for them to work with both C and C++ code:

- Rename any C- or C++-specific keywords. A listing of these can be found in the *Language Reference*.
- Remove any use of the `_Packed` keyword and replace it with the appropriate use of **#pragma** `pack`. C++ does not support `_Packed`.
- Remove any use of **#pragma** `linkage` and add the appropriate linkage convention keyword must be added to the prototype. C++ does not support **#pragma** `linkage` directives.
- Add to the beginning of each header file:

  ```
  #if __cplusplus
  extern "C" {
  #endif
  ```

- Add to the end of each header file:

  ```
  #if __cplusplus
  }
  #endif
  ```

Creating New Headers to Work with Both C and C++ (32-bit)

Follow these guidelines to enable your new header files to work with both C and C++ code:

- Rename any C- or C++-specific keywords. A listing of these can be found in the *Language Reference*.
- Add to the beginning of each header file:

  ```
  #if __cplusplus
  extern "C" {
  #endif
  ```

- Add to the end of each header file:

  ```
  #if __cplusplus
  }
  #endif
  ```

- Do not use `_Packed` in your code; use **#pragma** `pack` instead.
- Do not use **#pragma** `linkage` in your code; use the linkage convention keywords instead.

ANSI Notes

- Use `typedefs` for structures to be passed to 16-bit code and specify the `typedef` in a #**pragma** `seg16` directive.
- Specify the linkage on any identifiers that are pointers to functions.
- Use the **_Seg16** type qualifier to declare external pointers that will be shared between 32-bit and 16-bit code (and are declared in both). The **_Seg16** qualifier directs the compiler to store the pointer as a segmented pointer (with a 16-bit selector and 16-bit offset) that can be used directly by a 16-bit application. You can also use the pointer in a 32-bit program; VisualAge C++ compiler automatically converts it to a flat 32-bit pointer when necessary.

B VisualAge C++ Macros and Functions

This appendix lists the predefined macros reserved for use by the VisualAge C++ product. It also includes a list of the intrinsic and built-in functions. For a complete list of all functions in the VisualAge C++ runtime libraries, see the *C Library Reference*.

Predefined Macros

The macros identified in this section are provided to allow customers to write programs that use VisualAge C++ services. Only those macros identified in this section should be used to request or receive VisualAge C++ services.

VisualAge C++ compiler provides both the SAA predefined macros and a number of macros specific to VisualAge 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 C++ compiler, all language constructs are allowed. Defined using the **`#pragma langlvl(extended)`** directive or `/Se` compiler option.

Predefined Macros

VisualAge 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__	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_OS2__	Indicates that the host operating system is OS/2.
__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.
__MULTI__	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 /Gd option.
__OS2__	Set to the integer 1. Indicates the product is an OS/2 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.

`__THW_INTEL__`	Indicates that the target hardware is an Intel processor.
`__TOS_OS2__`	Indicates that the target operating system is OS/2.
`__TILED__`	Indicates tiled memory is being used. Defined using the `/Gt` compiler option.
`__32BIT__`	Set to the integer 1. Indicates the product is a 32-bit compiler.

The value of the `__IBMC__` and `__IBMCPP__` macros is 300. 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 `__OS2__`, `_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.

Intrinsic Functions

The VisualAge 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 C++ compiler by default inlines (generates code instead of a function call) the following C library functions:

abs	labs	memmove	strchr	strncat
_clear87	memchr	memset	strcmp	strncmp
_control87	memcmp	_status87	strcpy	strncpy
fabs	memcpy	strcat	strlen	strrchr

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.

You can override the inlining either by undefining the macro or by placing the name of the function in parentheses, thus disabling the processor 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 intrinsic functions.

Intrinsic Functions

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:

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

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.

Locale Categories

This appendix provides a listing of the categories which define a locale, the keywords used in each category, and the values which are valid for each keyword.

The following locate categories are described:

- LC_CTYPE Category
- LC_COLLATE Category
- LC_MONETARY Category
- LC_NUMERIC Category
- LC_TIME Category
- LC_MESSAGES Category
- LC_TOD Category
- LC_SYNTAX Category

LC_CTYPE Category

This category defines character classification, case conversion, and other character attributes. In this category, you can represent a series of characters by using three adjacent periods as an ellipsis symbol (`...`). An ellipsis is interpreted as including all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value following the ellipsis.

An ellipsis is valid within a single encoded character set.

For example, `\x30;...;\x39;` includes in the character class all characters with encoded values from `\x30` to `\x39`.

The keywords recognized in the `LC_CTYPE` category are listed below. In the descriptions, the term "automatically included" means that it is not an error either to include or omit any of the referenced characters; they are assumed by default even if the entire keyword is missing and accepted if present.

When a character is automatically included, it has an encoded value dependent on the `charmap` file in effect. If no `charmap` file is specified, the encoding of the encoded character set IBM-850 is assumed.

`copy` Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keywords are present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`upper` Defines characters to be classified as uppercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The uppercase letters `A` through `Z` are automatically included in this class.

The `isupper` and `iswupper` functions test for any character and wide character, respectively, included in this class.

`lower` Defines characters to be classified as lowercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The lowercase letters `a` through `z` are automatically included in this class.

The `islower` and `iswlower` functions test for any character and wide character, respectively, included in this class.

`alpha` Defines characters to be classified as letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. Characters classified as either `upper` or `lower` are automatically included in this class.

The `isalpha` and `iswalpha` functions test for any character or wide character, respectively, included in this class.

`digit` Defines characters to be classified as numeric digits. Only the digits `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`. can be specified. If they are, they must be in contiguous ascending sequence by numerical value. The digits `0` through `9` are automatically included in this class.

The `isdigit` and `iswdigit` functions test for any character or wide character, respectively, included in this class.

`space` Defines characters to be classified as whitespace characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, or `xdigit` can be specified for `space`. The characters `<space>, <form-feed>, <newline>, <carriage-return>`, `<horizontal-tab>`, and `<vertical-tab>`, and any characters defined in the class `blank` are automatically included in this class.

The functions `isspace` and `iswspace` test for any character or wide character, respectively, included in this class.

`cntrl` Defines characters to be classified as control characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `punct`, `graph`, `print`, or `xdigit` can be specified for `cntrl`.

The functions `iscntrl` and `iswcntrl` test for any character or wide character, respectively, included in this class.

`punct` Defines characters to be classified as punctuation characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `cntrl`, or `xdigit`, or as the `<space>` character, can be specified.

The functions `ispunct` and `iswpunct` test for any character or wide character, respectively, included in this class.

`graph` Defines characters to be classified as printing characters, not including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, and `punct` are automatically included. No character specified in the keyword `cntrl` can be specified for `graph.`

The functions `isgraph` and `iswgraph` test for any character or wide character, respectively, included in this class.

`print` Defines characters to be classified as printing characters, including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, `punct`, and the `<space>` character are automatically included. No character specified in the keyword `cntrl` can be specified for `print`.

The functions `isprint` and `iswprint` test for any character or wide character, respectively, included in this class.

`xdigit` Defines characters to be classified as hexadecimal digits. Only the characters defined for the class `digit` can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits `10` through `15`, with each set in ascending order (for example, `A, B, C, D, E, F, a, b, c, d, e, f`). The digits `0` through `9`, the uppercase letters `A` through `F`, and the lowercase letters `a` through `f` are automatically included in this class.

The functions `isxdigit` and `iswxdigit` test for any character or wide character, respectively, included in this class.

`blank` Defines characters to be classified as blank characters. The characters `<space>` and `<tab>` are automatically included in this class.

The functions `isblank` and `iswblank` test for any character or wide character, respectively, included in this class.

`toupper` Defines the mapping of lowercase letters to uppercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed in parentheses. The first character in each pair is the lowercase letter, and the second is the corresponding uppercase letter. Only characters specified for the keywords `lower` and `upper` can be specified for `toupper`. The lowercase letters `a` through `z`, their corresponding uppercase letters `A` through `Z`, are automatically in this mapping, but only when the toupper keyword is omitted from the locale definition.

It affects the behavior of the toupper and towupper functions for mapping characters and wide characters, respectively.

tolower Defines the mapping of uppercase letters to lowercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed by parentheses. The first character in each pair is the uppercase letter, and the second is its corresponding lowercase letter. Only characters specified for the keywords lower and upper can be specified. If the tolower keyword is omitted from the locale definition, the mapping is the reverse mapping of the one specified for the toupper.

The tolower keyword affects the behavior of the tolower and towlower functions for mapping characters and wide characters, respectively.

You may define additional character classes using your own keywords. A maximum of 32 classes are supported in total: the 12 standard classes, and up to 20 user-defined classes. The 12 standard classes being composed of the 11 standard classes listed above plus the ALNUM class which contains the total characters in the ALPHA and DIGIT classes.

The defined classes affect the behavior of wctype and iswctype functions.

Here is an example of the definition of the LC_CTYPE category:

```
#############
LC_CTYPE
#############
# upper letters are A-Z by default plus the three defined below
upper   <A-acute>;<A-grave>;<C-acute>

# lower case leters are a-z by default plus the three defined below
lower   <a-acute>;<a_grave><c-acute>

# space characters are default 6 characters plus the one defined below
space   <hyphen-minus>

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;<NUL>;\
        <SO>;<SI>

# default graph, print,punct, digit, xdigit, blank classes

# toupper mapping defined only for the following three pairs
toupper (<a-acute),<A-acute>);\
        (<a-grave),<A-grave>);\
        (<c-acute),<C-acute>);

# default upper to lower case mapping

# user defined class
myclass  <e-ogonek>;<E-ogonek>

END LC_CTYPE
```

LC_COLLATE Category

A collation sequence definition defines the relative order between collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values. It assigns each element one or more collation values (also known as collation weights). The collation sequence definition is used by regular expressions, pattern matching, and sorting and collating functions. The following capabilities are provided:

1. **Multicharacter collating elements.** Specification of multicharacter collating elements (sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements.** Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching, and

unless collation weights are explicitly specified, also as the collation weight to be used in sorting.

3. **Multiple weights and equivalence classes.** Collating elements can be assigned 1 to 6 collating weights for use in sorting. The first weight is referred to as the primary weight.
4. **One-to-many mapping.** A single character is mapped into a string of collating elements.
5. **Many-to-Many substitution.** A string of one or more characters are mapped to another string (or an empty string). The character or characters are ignored for collation purposes.
6. **Equivalence class definition.** Two or more collating elements have the same collation value (primary weight).
7. **Ordering by weights.** When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements. Each successive pair of elements is compared according to the relative primary weights for the elements. If they are equal, and more than one weight is assigned, then the pairs of collating elements are compared again according to the relative subsequent weights, until either two collating elements are not equal or the weights are exhausted.

Collating Rules

Collation rules consist of an ordered list of collating order statements, ordered from lowest to highest. The `<NULL>` character is considered lower than any other character. The ellipsis symbol (`"..."`) is a special collation order statement. It specifies that a sequence of characters collate according to their encoded character values. It causes all characters with values higher than the value of the `<collating identifier>` in the preceding line, and lower than the value for the `<collating identifier>` on the following line, to be placed in the character collation order between the previous and the following collation order statements in ascending order according to their encoded character values.

The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable among implementations.

The ellipsis symbol must be on a line by itself, *not* the first or last line, and the preceding and succeeding lines must not specify a weight.

A collating order statement describes how a collating identifier is weighted.

Each `<collating-identifier>` consists of a character, `<collating-element>`, `<collating-symbol>`, or the special symbol `UNDEFINED`. The order in which collating elements are specified determines the character order sequence, such that each collating element is considered lower than the elements following it. The `<NULL>` character is considered lower than any other character. Weights are expressed as characters, `<collating-symbol>`s, `<collating-element>`s, or the special symbol `IGNORE`. A single character, a `<collating-symbol>`, or a `<collating-element>` represents the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus rather than assigning absolute values to weights, a particular weight is expressed using the relative "order value" assigned to a collating element based on its order in the character collation sequence.

A `<collating-element>` specifies multicharacter collating elements, and indicates that the character sequence specified by the `<collating-element>` is to be collated as a unit and in the relative order specified by its place.

A `<collating-symbol>` can define a position in the relative order for use in weights. Do not use a `<collating-symbol>` to specify a weight.

The `<collating-symbol>` `UNDEFINED` is interpreted as including all characters not specified explicitly. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their encoded character values. If no `UNDEFINED` symbol is specified, and the current coded character set contains characters not specified in this clause, the LOCALDEF utility issues a warning and places such characters at the end of the character collation order.

The syntax for a collation order statement is:

```
<collating-identifier> <weight1>;<weight2>;...;<weightn>
```

Collation of two collating identifiers is done by comparing their relative primary weights. This process is repeated for successive weight levels until the two identifiers are different, or the weight levels are exhausted. The operands for each collating identifier define the primary, secondary, and subsequent relative weights for the collating identifier. Two or more collating elements can be assigned the same weight. If two collating identifiers have the same primary weight, they belong to the same *equivalence class*.

The special symbol `IGNORE` as a weight indicates that when strings are compared using the weights at the level where `IGNORE` is specified, the collating element should be ignored, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are `IGNORE`d in their primary weight form an equivalence class.

All characters specified by an ellipsis are assigned unique weights, equal to the relative order of the characters. Characters specified by an explicit or implicit

UNDEFINED special symbol are assigned the same primary weight (they belong to the same equivalence class).

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character "<ezset>" is given the string "<s><s>" as a weight, comparisons are performed as if all occurrences of the character <ezset> are replaced by <s><s> (assuming <s> has the collating weight <s>). If it is desirable to define <ezset> and <s><s> as an equivalence class, then a collating element must be defined for the string "ss".

If no weight is specified, the collating identifier is interpreted as itself.

For example, the order statement

```
<a>   <a>
```

is equivalent to

```
<a>
```

Collating Keywords

The following keywords are recognized in a collation sequence definition.

`copy`
: Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

`collating-element`
: Defines a collating-element symbol representing a multicharacter collating element. This keyword is optional.

  In addition to the collating elements in the character set, the collating-element keyword can be used to define multicharacter collating elements. The syntax is:

  "collating-element %s from %s\n", *<collating-element>*, *<string>*

  The <collating-element> should be a symbolic name enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collate as an entity. A <collating-element> defined with this keyword is only recognized within the LC_COLLATE category.

  For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

`collating-symbol`

Defines a collating symbol for use in collation order statements.

The `collating-symbol` keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight. This keyword is optional.

This construct can define symbols for use in collation sequence statements, between the `order_start` and `order_end` keywords.

The syntax is:

`"collating-symbol %s\n",` *`<collating-symbol>`*

The `<collating-symbol>` must be a symbolic name, enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current `charmap` file (if any), or any other symbolic name defined in this collation definition. A `<collating-symbol>` defined with this keyword is only recognized within the `LC_COLLATE` category.

For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

`substitute`

The `substitute` keyword defines a substring substitution in a string to be collated. This keyword is optional. The following operands are supported with the `substitute` keyword:

`"substitute %s with %s\n",` *`<regular-expr>`*`,` *`<replacement>`*

The first operand is treated as a basic regular expression. The replacement operand consists of zero or more characters and regular expression back-references (for example, `\1` through `\9`). The back-references consist of the backslash followed by a digit from 1 to 9. If the backslash is followed by two or three digits, it is interpreted as an octal constant.

When strings are collated according to a collation definition containing substitute statements, the collation behaves as if occurrences of substrings matching the basic regular expression are replaced by the replacement string, before the strings are compared based on the specified collation sequence. Ranges in the regular expression are interpreted according to the current character collation sequence and character classes according to the character classification specified by the `LC_CTYPE` environment variable at collation time. If more than one substitute statement is present in the collation definition, the collation process behaves as if the substitute statements are applied to the strings in the order they occur in the source definition. The substitution for the substitute statements are processed before any substitutions for one-to-many mappings. The support of the "substitute" keyword is an IBM VisualAge C++ extension to the POSIX standard.

`order_start`
: Define collating rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

The `order_start` keyword must precede collation order entries. It defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the `order_start` keyword is:

```
order_start <sort-rule1>;<sort-rule1>;...;<sort-rulen>
```

The operands of the `order_start` keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one `forward` operand is assumed. If any is present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (;). Each operand consists of one or more collation directives separated by commas (,). If the number of operands exceeds the limit of 6, the LOCALDEF utility issues a warning message.

The following directives are supported:

`forward`
: specifies that comparison operations for the weight level proceed from the start of the string towards its end.

`backward`
: specifies that comparison operations for the weight level proceed from the end of the string toward its beginning.

`no-substitute`
: no substitution is performed, such that the comparison is based on collation values for collating elements before any substitution operations are performed.

Notes:

1. This is an IBM VisualAge C++ extension to the POSIX standard.
2. When the `no-substitute` keyword is specified, one-to-many mappings are ignored.

`position`
: specifies that comparison operations for the weight level must consider the relative position of non-`IGNORE`d elements in the strings. The string containing a non-`IGNORE`d element after the fewest `IGNORE`d collating elements from the start of the comparison collates first. If both strings contain a non-`IGNORE`d character in the same relative position, the collating values assigned to the elements determine the order. If the strings are equal, subsequent non-`IGNORE`d characters are considered in the same manner.

`order_end`
: The collating order entries are terminated with an `order_end` keyword.

Here is an example of an `LC_COLLATE` category:

```
       LC_COLLATE
       # ARTIFICIAL COLLATE CATEGORY

       # collating elements
1      collating-element   <ch>  from "<c><h>"
       collating-element   <Ch>  from "<C><h>"
       collating-element   <eszet> from "<s><z>"

       %collating symbols for relative order definition

       collating-symbol    <LOW>
2      collating-symbol    <UPPER-CASE>
       collating-symbol    <LOWER-CASE>
       collating-symbol    <NONE>


3      order_start forward;backward;forward
       <NONE>
4      <LOW>
       <UPPER-CASE>
       <LOWER-CASE>

5      UNDEFINED IGNORE;IGNORE;IGNORE

       <space>
6      ...
       <quotation-mark>
7      <a>           <a>;<NONE>;<LOWER-CASE>
10     <a-acute>     <a>;<a-acute>;<LOWER-CASE>
11     <a-grave>     <a>;<a-grave>;<LOWER-CASE>
8      <A>           <a>;<NONE>;<UPPER-CASE>
11     <A-acute>     <a>;<a-acute>;<UPPER-CASE>
```

```
11        <A-grave>     <a>;<a-grave>;<UPPER-CASE>
11        <ch>          <ch>;<NONE>;<LOWER-CASE>
11        <Ch>          <ch>;<NONE>;<UPPER-CASE>
 9        <s>           <s>;<s>;<LOWER-CASE>
12        <eszet>       "<s><s>";"<eszet><s>";<LOWER-CASE>
 9        <z>           <z>;<NONE>;<LOWER-CASE>
          order_end
```

The example is interpreted as follows:

1 collating elements

- character `<c>` followed by `<h>` collate as one entity named `<ch>`
- character `<C>` followed by `<h>` collate as one entity named `<Ch>`
- character `<s>` followed by `<z>` collate as one entity named `<eszet>`

2 collating symbols `<LOW>`, `<UPPER-CASE>`, `<LOWER-CASE>` and `<NONE>` are defined to be used in relative order definition

3 up to 3 string comparisons are defined:

- first pass starts from the beginning of the strings
- second pass starts from the end of the strings, and
- third pass starts from the beginning of the strings

4 the collating weights are defined such that

- `<LOW>` collates before `<UPPER-CASE>`,
- `<UPPER-CASE>` collates before `<LOWER-CASE>`,
- `<LOWER-CASE>` collates before `<NONE>`;

5 all characters for which collation is not specified here are ordered after `<NONE>`, and before `<space>` in ascending order according to their encoded values

6 all characters with an encoded value larger than the encoded value of `<space>` and lower than the encoded value of `<quotation-mark>` in the current encoded character set, collate in ascending order according to their values;

7 `<a>` has a:

- primary weight of `<a>`,
- secondary weight `<NONE>`,
- tertiary weight of `<LOWER-CASE>`,

8 `<A>` has a:

- primary weight of `<a>`,
- secondary weight of `<NONE>`,
- tertiary weight of `<UPPER-CASE>`,

9 the weights of `<s>` and `<z>` are determined in a similar fashion to `<a>` and `<A>`.

10 `<a-acute>` has a:

- primary weight of `<a>`,
- secondary weight of `<a-acute>` itself,
- tertiary weight of `<LOWER-CASE>`,

11 the weights of `<a-grave>`, `<A-acute>`, `<A-grave>`, `<ch>` and `<Ch>` are determined in a similar fashion to `<a-acute>`.

12 `<eszet>` has a:

- primary weight determined by replacing each occurence of `<eszet>` with the sequence of two `<s>`'s and using the weight of `<s>`,
- secondary weight determined by replacing each occurence of `<eszet>` with the sequence of `<eszet>` and `<s>` and using their weights,
- tertiary weight is the relative position of `<LOWER-CASE>`.

Comparison of Strings

Compare the strings `s1="aAch"` and `s2="AaCh"` using the above `LC_COLLATE` definition:

1. `s1=> "aA<ch>"`, and `s2=> "Aa<Ch>"`
2. first pass:
   a. substitute the elements of the strings with their primary weights:

      `s1=> "<a><a><ch>", s2=> "<a><a><ch>"`

   b. compare the two strings starting with the first element — they are equal.
3. second pass:
   a. substitute the elements of the strings with their secondary weights:

      `s1=> "<NONE><NONE><NONE>", s2=>"<NONE><NONE><NONE>"`

   b. compare the two strings from the last element to the first — they are equal.
4. third pass:

   a. substitute the elements of the strings with their third level weights:

      `s1=> "<LOWER-CASE><UPPER-CASE><LOWER-CASE>", s2=> "<UPPER-CASE><LOWER-CASE><UPPER-CASE>",`

   b. compare the two strings starting from the beginning of the strings:

      `s2` compares lower than `s1`, because `<UPPER-CASE>` is before `<LOWER-CASE>`.

Compare the strings `s1="a1sz"` and `s2=>"a2ss"`:

1. `s1=> "a1<eszet>"` and `s2= "a2ss"`;
2. first pass:
   a. substitute the elements of the strings with their primary weights:

      `s1=> "<a><s><s>", s2=> "<a><s><s>"`

   b. compare the two strings starting with the first element — they are equal.
3. second pass:
   a. substitute the elements of the strings with their secondary weights:

      `s1=> "<a-acute><eszet><s>", s2=>"<a-grave><s><s>"`

   b. compare the two strings from the last element to the first — `<s>` is before `<ezset>`.

LC_MONETARY Category

This category defines the rules and symbols used to format monetary quantities. The operands are strings or integers. The following keywords are supported:

`copy`
: Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`int_curr_symbol`
: Specifies the international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO4217 *Codes for the Representation of Currency and Funds*. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

  If not defined, it defaults to the empty string ("").

`currency_symbol`
: Specifies the string used as the local currency symbol. If not defined, it defaults to the empty string ("").

`mon_decimal_point`
: The string used as a decimal delimiter to format monetary quantities. If not defined it defaults to the empty string ("").

`mon_thousands_sep`
: Specifies the string used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. If not defined, it defaults to the empty string ("").

`mon_grouping`
: Defines the size of each group of digits in formatted monetary quantities. The operand is a string representing a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is −1, then no further grouping is performed. If not defined, `mon_grouping` defaults to −1 which indicates that no grouping. An empty string is interpreted as −1.

`positive_sign`
: A string used to indicate a formatted monetary quantity with a non-negative value. If not defined, it defaults to the empty string ("").

`negative_sign`
: Specifies a string used to indicate a formatted monetary quantity with a negative value. If not defined, it defaults to the empty string ("").

`int_frac_digits`
: Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `int_curr_symbol`. If not defined, it defaults to −1.

`frac_digits`
: Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `currency_symbol`. If not defined, it defaults to −1.

`p_cs_precedes`
: Specifies an integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a non-negative formatted monetary quantity, and set to `0` if the symbol succeeds the value. If not defined, it defaults to −1.

p_sep_by_space
: Specifies an integer set to 0 if no space separates the currency_symbol or int_curr_symbol from the value for a non-negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to −1.

n_cs_precedes
: An integer set to 1 if the currency_symbol or int_curr_symbol precedes the value for a negative formatted monetary quantity, and set to 0 if the symbol succeeds the value. If not defined, it defaults to −1.

n_sep_by_space
: An integer set to 0 if no space separates the currency_symbol or int_curr_symbol from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to −1.

p_sign_posn
: An integer set to a value indicating the positioning of the positive_sign for a non-negative formatted monetary quantity. The following integer values are recognized:

0 Parentheses surround the quantity and the currency_symbol or int_curr_symbol.
1 The sign string precedes the quantity and the currency_symbol or int_curr_symbol.
2 The sign string succeeds the quantity and the currency_symbol or int_curr_symbol.
3 The sign string immediately precedes the currency_symbol or int_curr_symbol.
4 The sign string immediately succeeds the currency_symbol or int_curr_symbol.

The following value may also be specified, though it is not part of the POSIX standard.

5 Use debit-sign or credit-sign for p_sign_posn or n_sign_posn.

If not defined, it defaults to −1.

n_sign_posn
: An integer set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity. The recognized values are the same as for p_sign_posn. If not defined, it defaults to −1.

`left_parenthesis`
: The symbol of the locale's equivalent of `(` to form a negative-valued formatted monetary quantity together with `right_parenthesis`. If not defined, it defaults to the the empty string ("").

  Note: This is an IBM-specific extension.

`right_parenthesis`
: The symbol of the locale's equivalent of `)` to form a negative-valued formatted monetary quantity together with `left_parenthesis`. If not defined, it defaults to the the empty string ("");

  Note: This is an IBM-specific extension.

`debit_sign`
: The symbol of locale's equivalent of `DB` to indicate a non-negative-valued formatted monetary quantity. If not defined, it defaults to the the empty string ("");

  Note: This is an IBM-specific extension.

`credit_sign`
: The symbol of locale's equivalent of `CR` to indicate a negative-valued formatted monetary quantity. If not defined, it defaults to the the empty string ("");

  Note: This is an IBM-specific extension.

Here is an example of the definition of the `LC_MONETARY` category:

```
#############
LC_MONETARY
#############

int_curr_symbol   "<J><P><Y><space>"
currency_symbol   "<yen>"
mon_decimal_point "<period>"
mon_thousands_sep "<comma>"
mon_grouping      "3;0"
positive_sign     ""
negative_sign     "<hyphen-minus>"
int_frac_digits   0
frac_digits       0
p_cs_precedes     1
p_sep_by_space    0
n_cs_precedes     1
n_sep_by_space    0
p_sign_posn       1
n_sign_posn       1
debit_sign        "<D><B>"
credit_sign       "<C><R>"
```

```
left_parenthesis  "<left-parenthesis>"
right_parenthesis "<right-parenthesis>"

END LC_MONETARY
```

LC_NUMERIC Category

This category defines the rules and symbols used to format non-monetary numeric information. The operands are strings. The following keywords are recognized:

`copy`
: Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`decimal_point`
: Specifies a string used as the decimal delimiter in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string.

`thousands_sep`
: Specifies a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary, formatted quantities.

`grouping`
: Defines the size of each group of digits in formatted non-monetary quantities. The operand is a string representing a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is −1, then no further grouping is performed. An empty string is interpreted as −1.

Here is an example of how to specify the `LC_NUMERIC` category:

```
#############
LC_NUMERIC
#############

decimal_point     "<comma>"
thousands_sep     "<space>"
grouping          "3;0"

END LC_NUMERIC
```

LC_TIME Category

The `LC_TIME` category defines the interpretation of the field descriptors used for parsing, then formatting, the date and time. Refer to the `strftime` and `strptime` functions in the *C Library Reference* for a description of format specifiers.

The following keywords are supported:

`copy`
: Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`abday`
: Defines the abbreviated weekday names, corresponding to the `%a` field descriptor. The operand consists of seven semicolon-separated strings. The first string is the abbreviated name corresponding to Sunday, the second string corresponds to Monday, and so forth.

`day`
: Defines the full weekday names, corresponding to the `%A` field descriptor. The operand consists of seven semicolon-separated strings. The first string is the full name corresponding to Sunday, the second string to Monday, and so forth.

`abmon`
: Defines the abbreviated month names, corresponding to the `%b` field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

`mon`
: Defines the full month names, corresponding to the `%B` field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

`d_t_fmt`
: Defines the appropriate date and time representation, corresponding to the `%c` field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

`d_fmt`
: Defines the appropriate date representation, corresponding to the `%x` field descriptor. The operand consists of a string, and may contain any combination of characters and field descriptors.

`t_fmt`
: Defines the appropriate time representation, corresponding to the `%X` field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

`am_pm`
: Defines the appropriate representation of the ante meridian and post meridian strings, corresponding to the `%p` field descriptor. The operand consists of two strings, separated by a semicolon. The first string represents the ante meridian designation, the last string the post meridian designation.

`t_fmt_ampm`
: Defines the appropriate time representation in the 12-hour clock format with am_pm, corresponding to the `%r` field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors.

`era`
: Defines how the years are counted and displayed for each era (or emperor's reign) in a locale.

No `era` is needed if the `%E` field descriptor modifier is not used for the locale.

For each `era`, there must be one string in the following format:

`direction:offset:start_date:end_date:name:format`

where

`direction`
: Either a + or – character. The + character indicates the time axis should be such that the years count in the positive direction when moving from the starting date towards the ending date. The – character indicates the time axis should be such that the years count in the negative direction when moving from the starting date towards the ending date.

`offset`
: A number of the first year of the era.

`start_date`
: A date in the form yyyy/mm/dd where yyyy, mm and dd are the year, month and day numbers, respectively, of the start of the era. Years prior to the year AD `0` are represented as negative numbers. For example, an era beginning March 5th in the year `100` BC would be represented as `-100/3/5`.

`end_date`
: The ending date of the era in the same form as the start_date above or one of the two special values –* or +*. A value of –* indicates the ending date of the era extends to the beginning of time while +* indicates it extends to the end of time. The ending date may be either before or after the starting

date of an era. For example, the strings for the Christian eras AD and BC would be:

```
+:0:0000/01/01:+*:AD:%EC %Ey
+:1:-0001/12/31:-*:BC:%EC %Ey
```

`name`
A string representing the name of the era which is substituted for the `%EC` field descriptor.

`format`
A string for formatting the `%EY` field descriptor. This string is usually a function of the `%EC` and `%Ey` field descriptors.

The operand consists of one string for each `era`. If there is more than one `era`, strings are separated by semicolons.

`era_year`
Defines the format of the year in alternate `era` format, corresponding to the `%EY` field descriptor.

`era_d_fmt`
Defines the format of the date in alternate `era` notation, corresponding to the `%Ex` field descriptor.

`alt_digits`
Defines alternate symbols for digits, corresponding to the `%O` field descriptor modifier. The operand consists of semicolon-separated strings. The first string is the alternate symbol corresponding to zero, the second string the symbol corresponding to one, and so forth. A maximum of `100` alternate strings may be specified. The `%O` modifier indicates that the string corresponding to the value specified by the field descriptor is used instead of the value.

LC_MESSAGES Category

The `LC_MESSAGES` category defines the format and values for positive and negative responses.

The following keywords are recognized:

`copy`
Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`yesexpr`
The operand consists of an extended regular expression that describes the acceptable **affirmative** response to a question that expects an affirmative or negative response.

`noexpr`
The operand consists of an extended regular expression that describes the acceptable **negative** response to a question that expects an affirmative or negative response.

Here is an example that shows how to define the LC_MESSAGES category:

```
#############
LC_MESSAGES
#############
# yes expression is a string that starts with
# "SI", "Si" "sI" "si" "s" or "S"
yesexpr "<circumflex><left-parenthesis><left-square-bracket><s><S>\
<right-square-bracket><left-square-bracket><i><I><right-square-bracket>\
<vertical-line><left-square-bracket><s><S><right-square-bracket>\
<right-parenthesis>"

# no expression is a string that starts with
# "NO", "No" "nO" "no" "N" or "n"
noexpr "<circumflex><left-parenthesis><left-square-bracket><n><N>\
<right-square-bracket><left-square-bracket><o><O><right-square-bracket>\
<vertical-line><left-square-bracket><n><N><right-square-bracket>\
<right-parenthesis>"

END LC_MESSAGES
```

LC_TOD Category

The LC_TOD category defines the rules used to define the beginning, end, and duration of daylight savings time, and the difference between local time and Greenwich Mean time. This is an IBM extension.

The following keywords are recognized:

`copy`
Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

`timezone_difference`
: An integer specifying the time zone difference expressed in minutes. If the local time zone is west of the Greenwich Meridian, this value must be positive. If the local time zone is east of the Greenwich Meridian, this value must be negative. An absolute value greater than `1440` (the number of minutes in a day) for this keyword indicates that the run time library is to get the time zone difference from the system.

`timezone_name`
: A string specifying the time zone name such as `"PST"` (Pacific Standard Time) specified within quotation marks. The default for this field is a `NULL` string.

`daylight_name`
: A string specifying the Daylight Saving Time zone name, such as `"PDT"` (Pacific Daylight Time), if there is one available. The string must be specified within quotation marks. If DST information is not available, this is set to `NULL`, which is also the default. This field must be filled in if DST information as provided by the other fields is to be taken into account by the `mktime` and `localtime` functions. These functions ignore DST if this field is `NULL`.

`start_month`
: An integer specifying the month of the year when Daylight Saving Time comes into effect. This value ranges from 1 through 12 inclusive, with 1 corresponding to January and 12 corresponding to December. If DST is not applicable to a locale, `start_month` is set to `0`, which is also the default.

`end_month`
: An integer specifying the month of the year when Daylight Saving Time ceases to be in effect. The specifications are similar to those for `start_month`.

`start_week`
: An integer specifying the week of the month when DST comes into effect. Acceptable values range from `-4` to `+4`. A value of `4` means the fourth week of the month, while a value of `-4` means fourth week of the month, counting from the end of the month. Sunday is considered to be the start of the week. If DST is not applicable to a locale, `start_week` is set to `0`, which is also the default.

`end_week`
: An integer specifying the week of the month when DST ceases to be in effect. The specifications are similar to those for `start_week`.

: **Note:** The `start_week` and `end_week` need not be used. The `start_day` and `end_day` fields can specify either the day of the week or the day of the month. If day of month is specified, `start_week` and `end_week` become redundant.

`start_day`
: An integer specifying the day of the week or the day of the month when DST comes into effect. The value depends on the value of `start_week`. If

start_week is not equal to 0, this is the day of the week when DST comes into effect. It ranges from 0 through 6 inclusive, with 0 corresponding to Sunday and 6 corresponding to Saturday. If start_week equals 0, start_day is the day of the month (for the current year) when DST comes into effect. It ranges from 1 through to the last day of the month inclusive. The last day of the month is 31 for January, March, May, July, August, October, and December. It is 30 for April, June, September, and November. For February, it is 28 on non-leap years and 29 on leap years. If DST is not applicable to a locale, start_day is set to 0, which is also the default.

end_day
: An integer specifying the day of the week or the day of the month when DST ceases to be in effect. The specifications are similar to those for start_day.

start_time
: An integer specifying the number of seconds after 12:00 midnight, local standard time, when DST comes into effect. For example, if DST is to start at 2:am, start_time is assigned the value 7200; for 12:00 am (midnight), start_time is 0; for 1:00 am, it is 3600.

end_time
: An integer specifying the number of seconds after 12 midnight, local standard time, when DST ceases to be in effect. The specifications are similar to those for start_time.

shift
: An integer specifying the DST time shift, expressed in seconds. The default is 3600, for 1 hour.

uctname
: A string specifying the name to be used for Coordinated Universal Time. If this keyword is not specified, the uctname will default to "UTC".

Here is an example of how to define the LC_TOD category:

```
#############
LC_TOD
#############
# the time zone difference is 8hrs; the name of the daylight saving
# time is PDT, and it starts on the first Sunday of April at
# 2:00AM and ends on the second Sunday of October at
# 2:00AM
timezone_difference +480
timezone_name       "<P><S><T>"
daylight_name       "<P><D><T>"
start_month         4
end_month           10
start_week          1
end_week            2
start_day           1
end_day             30
start_time          7200
end_time            3600
shift               3600
END LC_TOD
```

LC_SYNTAX Category

The `LC_SYNTAX` category defines the variant characters from the portable character set. `LC_SYNTAX` is an IBM-specific extension. This category can be queried by the C library function `getsyntx` to determine the encoding of a variant character if needed.

Warning: Customizing the `LC_SYNTAX` category is not recommended. You should use the `LC_SYNTAX` values obtained from the `charmap` file when you use the LOCALDEF utility.

The operands for the characters in the `LC_SYNTAX` category accept the single byte character specification in the form of a symbolic name, the character itself, or the decimal, octal, or hexadecimal constant. The characters must be specified in the `LC_CTYPE` category as a *`punct`* character. The values for the `LC_SYNTAX` characters must be unique. If symbolic names are used to define the encoding, only the symbolic names listed for each character should be used.

The code points for the `LC_SYNTAX` characters are set to the code points specified. Otherwise, they default to the code points for the respective characters from the `charmap` file, if the file is present, or to the code points of the respective characters in the IBM-850 code page.

The following keywords are recognized:

Locale Categories

`copy`
: Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present.

  If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`backslash`
: Specifies a string that defines the value used to represent the backslash character. If this keyword is not specified, the value from the `charmap` file for the character `<backslash>`, `<reverse-solidus>`, or `<SM07>` is used, if it is present.

`right_brace`
: Specifies a string that defines the value used to represent the right brace character. If this keyword is not specified, the value from the `charmap` file for the character `<right-brace>`, `<right-curly-bracket>`, or `<SM14>` is used, if it is present.

`left_brace`
: Specifies a string that defines the value used to represent the left brace character. If this keyword is not specified, the value from the `charmap` file for the character `<left-brace>`, `<left-curly-bracket>`, or `<SM11>` is used, if it is present.

`right_bracket`
: Specifies a string that defines the value used to represent the right bracket character. If this keyword is not specified, the value from the `charmap` file for the character `<right-square-bracket>`, or `<SM08>` is used, if it is present.

`left_bracket`
: Specifies a string that defines the value used to represent the left bracket character. If this keyword is not specified, the value from the `charmap` file for the character `<left-square-bracket>`, or `<SM06>` is used, if it is present.

`circumflex`
: Specifies a string that defines the value used to represent the circumflex character. If this keyword is not specified, the value from the `charmap` file for the character `<circumflex>`, `<circumflex-accent>`, or `<SD15>` is used, if it is present.

`tilde`
: Specifies a string that defines the value used to represent the tilde character. If this keyword is not specified, the value from the `charmap` file for the character `<tilde>`, or `<SD19>` is used, if it is present.

`exclamation_mark`
: Specifies a string that defines the value used to represent the exclamation mark character. If this keyword is not specified, the value from the `charmap` file for the character `<exclamation-mark>`, or `<SP02>` is used, if it is present.

`number_sign`
: Specifies a string that defines the value used to represent the number sign character. If this keyword is not specified, the value from the `charmap` file for the character `<number-sign>`, or `<SM01>` is used, if it is present.

`vertical_line`
: Specifies a string that defines the value used to represent the vertical line character. If this keyword is not specified, the value from the `charmap` file for the character `<vertical-line>`, or `<SM13>` is used, if it is present.

`dollar_sign`
: Specifies a string that defines the value used to represent the dollar sign character. If this keyword is not specified, the value from the `charmap` file for the character `<dollar-sign>`, or `<SC03>` is used, if it is present.

`commercial_at`
: Specifies a string that defines the value used to represent the commercial at character. If this keyword is not specified, the value from the `charmap` file for the character `<commercial-at>`, or `<SM05>` is used, if it is present.

`grave_accent`
: Specifies a string that defines the value used to represent the grave accent character. If this keyword is not specified, the value from the `charmap` file for the character `<grave-accent>`, or `<SD13>` is used, if it is present.

Here is an example of how the `LC_SYNTAX` category is defined:

```
#############
LC_SYNTAX
#############

backslash        "<backslash>"
right_brace      "<right-brace>"
left_brace       "<left-brace>"
right_bracket    "<right-square-bracket>"
left_bracket     "<left-square-bracket>"
circumflex       "<circumflex>"
tilde            "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign      "<number-sign>"
vertical_line    "<vertical-line>"
dollar_sign      "<dollar-sign>"
commercial_at    "<commercial-at>"
grave_accent     "<grave-accent>"

END LC_SYNTAX
```

Regular Expressions

`Regular Expressions` (REs) are used to determine if a character string of interest is matched somewhere in a set of character strings. You can specify more than one character string for which you wish to determine if a match exists. Regular Expressions use collating values from the current locale definition file in the matching process.

The search for a matching sequence starts at the beginning of the string and stops when the first sequence matching the expression is found. The first sequence is the one that begins earliest in the string. If the pattern permits matching several sequences at this starting point, the longest sequence is matched.

To use a regular expression, first compile it with `regcomp`. You can then use `regexec` to compare the compiled expression to other expressions. If an error occurs, `regerror` provides information about the error. When you have finished with the expression, use `regfree` to free it from memory. All of these functions are described in more detail in the *C Library Reference*.

Basic Matching Rules

Within an RE:

- An ordinary character matches itself. The simplest form of regular expression is a string of characters with no special meaning.
- A special character preceded by a backslash matches itself. For basic regular expressions (BREs), the special characters are:

  ```
  . [ \ * ^ $
  ```

  For extended regular expressions (EREs), the special characters also include:

  ```
  ( ) + ? { |
  ```

  (EREs are supported when you specify the REG_EXTENDED flag.)
- A period (.) without a backslash matches any single character. For EREs, it matches any character except the null character.
- An expression within square brackets ([ ]), called a *bracket expression*, matches one or more characters or collating elements.

Note: Do not use multibyte characters in regular expressions.

Regular Expressions

Bracket Expressions

A bracket expression itself contains one or more expressions that represent characters, collating symbols, equivalence or character classes, or range expressions:

[*string*]
: Matches any of the characters specified. For example, `[abc]` matches any of `a`, `b`, or `c`.

[^*string*]
: Does **not** match any of the characters in *string*. The caret immediately following the left bracket ([) negates the characters that follow. For example, `[^abc]` matches any character or collating element **except** `a`, `b`, or `c`.

[*collat_sym–collat_sym*]
: Matches any collating elements that fall between the two specified collating symbols, inclusive. The two symbols must be different, and the second symbol must collate equal to or higher than the first. For example, in the "C" locale, `[r–t]` would match any of `r`, `s`, or `t`.

  Note: To treat the hyphen (–) as itself, place it either first or last in the bracket expression, for example: `[–rt]` or `[rt–]`. Both of these expressions would match `-`, `r`, or `t`.

[[.*collat_symbl*.]]
: Matches the collating element represented by the specified single or multicharacter collating symbol *collat_symbl*. For example, assuming that <ch> is the collating symbol for `ch` in the current locale, `[[.ch.]]` matches the character sequence `ch`. (In contrast, `[ch]` matches `c` or `h`.) If *collat_symbl* is not a collating element in the current locale, or if it has no characters associated with it, it is treated as an invalid expression.

[[=*collat_symbl*=]]
: Matches all collating elements that have a weight equivalent to the specified single or multicharacter collating symbol *collat_symbl*. For example, assuming `a`, `à`, and `â` belong to the same equivalence class, `[[=a=]]` matches any of the three. If the collating symbol does not have any equivalents, it is treated as a collating symbol and matches its corresponding collating element (as for [..]).

[[:*char_class*:]]
: Matches any characters that belong to the specified character class *char_class*. For example, `[[:alnum:]]` matches all alphanumeric characters (characters for which `isalnum` would return nonzero).

Note: To use the right bracket (]) in a bracket expression, you must specify it immediately following the left bracket ([) or caret symbol (^). For example, `[]x]` matches the characters] and x; `[^]x]` does not match] or x; `[x]]` is not valid.

You can combine characters, special characters, and bracket expressions to form REs that match multiple characters and subexpressions. When you concatenate the characters and expressions, the resulting RE matches any string that matches each component within the RE. For example, `cd` matches characters 3 and 4 of the string `abcde`; `ab[[:digit:]]` matches `ab3` but not `abc`. For EREs, you can optionally enclose the concatenation in parentheses.

Additional Syntax Specifiers

You can also use other syntax within an RE to control what it matches:

\\(*expression*\\)
: Matches whatever *expression* matches. You only need to enclose an expression in these delimiters to use operators (such as * or +) on it and to denote subexpressions for backreferencing (explained later in this section). For EREs, use the parentheses without the backslashes: (*subexpression*)

\n
: Matches the same string that was matched by the *n*th preceding expression enclosed in \\(\\) or, for EREs, ( ). This is called a *backreference*. *n* can be 1 through 9. For example, `\(ab\)\1` matches `abab`, but does not match `ac`. If fewer than *n* subexpressions precede \n, the backreference is not valid.

  Note: You cannot use backreferences in EREs.

*expression**
: Matches zero or more consecutive occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a backreference (for BREs). For example, `[ab]*` matches `ab` and `ababab`; `b*cd` matches characters 3 to 7 of `cabbbcdeb`.

expression\\{*m*\\}
: Matches exactly *m* occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a backreference (for BREs). For example, `c\{3\}` matches characters 5 through 7 of `ababccccd` (the first 3 `c` characters only). For EREs, use the braces without the backslashes: {*m*}

expression\\{m,\\}
: Matches at least *m* occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a backreference (for BREs). For example, `\(ab\)\{3,\}` matches `abababab`,

but does not match `ababac`. For EREs, use the braces without the backslashes: {*m*, }

expression\{m,u\}
: Matches any number of occurrences, between *m* and *u* inclusive, of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a backreference (for BREs). For example, `bc\{1,3\}` matches characters 2 through 4 of `abccd` and characters 3 through 6 of `abbcccccd` For EREs, use the braces without the backslashes: {*m*,*u*}

^*expression*
: Matches only sequences that match *expression* that start at the first character of a string or after a new-line character if the REG_NEWLINE flag was specified. For example, `^ab` matches `ab` in the string `abcdef`, but does **not** match it in the string `cdefab`. The *expression* can be the entire RE or any subexpression of it.

  Portability Note: When ^ is the first character of a subexpression, other implemenations could interpret it as a literal character. To ensure portability, avoid using ^ at the beginning of a subexpression; to use it as a literal character, precede it with a backslash.

expression$
: Matches only sequences that match *expression* that end the string or that precede the new-line character if the REG_NEWLINE flag was specified. For example, `ab$` matches `ab` in the string `cdefab` but does **not** match it in the string `abcdef`. The *expression* must be the entire RE.

  Portability Note: When $ is the last character of a subexpression, it is treated as a literal character. Other implementations could interpret is as described above. To ensure portability, avoid using $ at the end of a subexpression; to use it as a literal character, precede it with a backslash.

^*expression*$
: Matches only an entire string, or an entire line if the REG_NEWLINE flag was specified. For example, `^abcde$` matches only `abcde`.

In addition to those listed above, you can also use the following specifiers for EREs (they are not valid for BREs):

expression+
: Matches what one or more occurrences of *expression* matches. For example, `a+(bc)` matches `aaaaabc`; `(bc)+` matches characters 1 through 6 of `bcbcbcbb`.

expression?
: Matches zero or one consecutive occurrences of what *expression* matches. For example, `b?c` matches character 2 of `acabbb` (zero occurrences of `b` followed by `c`).

expression|*expression*
: Matches a string that matches either *expression*. For example, `a((bc)|d)` matches both `abd` and `ad`.

Order of precedence

Like C and C++ operators, the RE syntax specifiers are processed in a specific order. The order of precedence for BREs is described below, from highest to lowest. The specifiers in each category are also listed in order of precedence.

Collation-related bracket symbols	[==] [::] [..]
Special characters	\spec_char
Bracket expressions	[ ]
Subexpressions and backreferences	\\(\\) \\n
Repetition	* \\{m\\} \\{m,\\} \\{m,n\\}
Concatenation	
Anchoring	^ $

The order of precedence for EREs is:

Collation-related bracket symbols	[==] [::] [..]
Special characters	\spec_char
Bracket expressions	[ ]
Grouping	( )
Repetition	* + ? {m} {m,} {m,n}
Concetenation	
Anchoring	^ $
Alternation	|

E Mapping

This appendix describes how VisualAge C++ compiler maps data types into storage and the alignment of each data type and the mapping of its bits. The mapping of identifier names is also discussed, as is the encoding scheme used by the compiler for encoding or *mangling* C++ function names.

Name Mapping

To prevent conflicts between user-defined identifiers (variable names or functions) and VisualAge C++ library functions, do not use the name of any library function or external variable defined in the library as a user-defined function.

If you statically link to the VisualAge C++ runtime libraries (using the `/Gd-` option), all external names beginning with `Dos`, `Vio`, or `Kbd` (in the case given) become reserved external identifiers. These names are not reserved if you dynamically link to the libraries.

To prevent conflicts with internal names, do not use an underscore at the start of any of your external names; these identifiers are reserved for use by the compiler and libraries. The internal VisualAge C++ identifier names that are not listed in either the *Language Reference* or this manual all begin with an underscore (_).

If you have an application that uses a restricted name as an identifier, change your code or use a macro to globally redefine the name and avoid conflicts. You can also use the **`#pragma map`** directive to convert the name, but this directive is not portable outside of SAA.

A number of functions and variables that existed in the IBM C/2 and Microsoft C Version 6.0 compilers are implemented in the VisualAge C++ product, but with a preceding underscore to conform to ANSI naming requirements. When you run the VisualAge C++ compiler in extended mode (which is the default) and include the appropriate library header file, the original names are mapped to the new names for you. For example, the function name `putenv` is mapped to `_putenv`. When you compile in any other mode, this mapping does not take place.

Note: Because the name `timezone` is used as a structure field by the OS/2 operating system, the variable `_timezone` is **not** mapped to `timezone`.

Demangling (Decoding) C++ Function Names

When the VisualAge C++ compiler compiles a program, it encodes all function names and certain other identifiers to include type and scoping information. This encoding process is called *mangling*. The linker uses the mangled names to ensure type-safe linkage. These mangled names are used in the object files and in the final executable file. Tools that use these files must use the mangled names and not the original names used in the source code.

VisualAge C++ provides two methods of converting mangled names to the original source code names, demangling functions and the CPPFILT utility.

Using the Demangling Functions

The runtime library contains a small class hierarchy of functions that you can use to demangle names and examine the resulting parts of the name. It also provides a C-language interface you can use in C programs. The functions use no external C++ features.

The demangling functions are available in both the static (.LIB) and dynamic (.DLL) versions of the library. The interface is documented in the **<demangle.h>** header file.

Using the demangling functions, you can write programs to convert a mangled name to a demangled name and to determine characteristics of that name, such as its type qualifiers or scope. For example, given the mangled name of a function, the program returns the demangled name of the function and the names of its qualifiers. If the mangled name refers to a class member, you can determine if it is `static`, `const`, or `volatile`. You can also get the whole text of the mangled name.

To demangle a name, which is represented as a character array, create a dynamic instance of the `Name` class and provide the character string to the class's constructor. For example, to demangle the name `f__1XFi`, create:

```
char *rest;
Name *name = Demangle("f__1XFi", rest);
```

The demangling functions classify names into five categories: function names, member function names, special names, class names, and member variable names. After you construct an instance of class `Name`, you can use the `Kind` member function of `Name` to determine what kind of `Name` the instance is. Based on the kind of name returned, you can ask for the text of the different parts of the name or of the entire name.

For the mangled name `f__1XFi`, you can determine:

```
name->Kind() == MemberFunction
((MemberFunctionName *) name)->Scope()->Text() is "X"
((MemberFunctionName *) name)->RootName() is "f"
((MemberFunctionName *) name)->Text() is "X::f(int)"
```

If the character string passed to the `Name` constructor is not a mangled name, the `Demangle` function returns `NULL`.

For further details about the demangling functions and their C++ and C interfaces, refer to the information contained in the **<demangle.h>** header file. If you installed using the defaults, this header file should be in the INCLUDE directory under the main VisualAge C++ installation directory.

Using the CPPFILT Utility

The VisualAge C++ product also provides the CPPFILT utility to convert mangled names to demangled names. You can use this utility with:

- An ASCII text file to substitute demangled names for any mangled names found in the text.
- A binary (object or library) file to produce a list of demangled names including exported, public, and referenced symbol names.

All CPPFILT output is sent to **stdout**. You can use the standard OS/2 redirection symbols to redirect the output to a file.

One of the applications of this utility is creating module definition files for your C++ DLLs. Because functions in the DLL have mangled names, when you list the `EXPORTS` in your `.DEF`, you must use the mangled names. You can use the CPPFILT utility to extract all the names from the object module for you, copy the ones you want to export into your `.DEF` file, and link your object module into a DLL.

Demangling C++ Function Names

The CPPFILT syntax is:

```
           ┌──────────────┐  ┌──────────────┐
►►──CPPFILT──▼──────────────┴──▼──────────────┴──────────────────────►◄
             └─/option─┘        └─filename─┘
```

where *option* is one or more CPPFILT options and *filename* is the name of the file containing the mangled names. If you do not specify a *filename*, CPPFILT reads the input from **stdin**. If you specify the /B option to run CPPFILT in binary mode, you must specify a *filename*. The file specified must be in the current directory unless you specify the full path name. CPPFILT will also search for library files along the paths specified in the LIB environment variable if it cannot find them in the current directory.

You can specify options in upper- or lowercase and precede them with either a slash (for example, /B) or a dash (-B). By default, all options are off.

Three options apply to both text and binary files:

/H or /?	Display online help on the CPPFILT command syntax and options.
/Q	Suppress the logo and copyright notice display.
/S	Demangle compiler-generated symbol names.

The following options apply only to text files:

/C	Demangle stand-alone class names, meaning names that do not appear within the context of a function name or member variable. The compiler does not usually produce these names. For example, Q2_1X1Y would be demangled as X::Y if you specify /C. Otherwise it is not demangled.
/M	Produce a symbol map containing a list of the mangled names and the corresponding demangled names. The symbol map is displayed after the usual CPPFILT output.
/T	Replace each mangled name in the text with its demangled name followed by the mangled name. (The default is to replace the mangled name with the demangled name only.)
/W *width*	Set the width of the field for demangled names in the output to *width* characters. If a demangled name is shorter than *width*, it is padded to the right with blanks; if longer, it is truncated to *width* characters. If you do not specify the /W option, there is no fixed width for the field.

The following options apply only to binary (object and library) files:

/B Run in binary mode. If you do not explicitly specify this option, CPPFILT runs in text mode.

/N Generate the NONAME keyword, used in EXPORTS statements in module definition files, to indicate that the exported names should be referenced by their ordinal numbers only and not by name. Use this option with the /O option.

/O [*ord*] Generate an ordinal number for each demangled name. You can optionally specify the ordinal number, *ord*, that CPPFILT should use as the first number. The ordinals are generated with the @ symbol and are consistent with the module definition file syntax. For example, if you specify /O 1000, the output for the first name might look like:

```
;ILinkedSequenceImpl::isConsistent() const
isConsistent__19ILinkedSequenceImplCFv    @1000
```

If you specify /O 1000 /N, the output for the same name would be:

```
;ILinkedSequenceImpl::isConsistent() const
isConsistent__19ILinkedSequenceImplCFv    @1000 NONAME
```

/P Include all public (COMDAT, COMDEF, or PUBDEF) symbols in the output. Note that if a COMDAT symbol occurs more than once, only the first occurrence is included in the output. Subsequent occurrences of the symbol appear in the output as comments.

/R Include all referenced (EXTDEF) symbols in the output.

/X Include all exported (EXPDEF) symbols in the output.

Note: If you do not specify any of /P, /R, or /X options in binary mode, the output includes only the demangled library and object names without any symbol names.

Demangling C++ Function Names

For example, given the command:

```
CPPFILT /B /P /O 1000 /N C:\IBMCPP\LIB\DDE4CC.LIB
```

CPPFILT would produce output like the following:

```
;From library:  c:\ibmcpp\lib\dde4cc.lib
 ;From object file:  C:\ibmcpp\src\IILNSEQ.C
  ;PUBDEFs (Symbols available from object file):
   ;ILinkedSequenceImpl::setToPrevious(ILinkedSequenceImpl::Node*&) const
   setToPrevious__19ILinkedSequenceImplCFRPQ2_19ILinkedSequenceImpl4Node    @1000 NONAME
   ;ILinkedSequenceImpl::allElementsDo(void*,void*) const
   allElementsDo__19ILinkedSequenceImplCFPvT1    @1001 NONAME
   ;ILinkedSequenceImpl::isConsistent() const
   isConsistent__19ILinkedSequenceImplCFv    @1002 NONAME
   ;ILinkedSequenceImpl::setToNext(ILinkedSequenceImpl::Node*&) const
   setToNext__19ILinkedSequenceImplCFRPQ2_19ILinkedSequenceImpl4Node    @1003 NONAME
   ;ILinkedSequenceImpl::addAsNext(ILinkedSequenceImpl::Node*, ILinkedSequenceImpl::Node*)
   addAsNext__19ILinkedSequenceImplFPQ2_19ILinkedSequenceImpl4NodeT1    @1004 NONAME
 ;From object file:  C:\ibmcpp\src\IITBSEQ.C
  ;PUBDEFs (Symbols available from object file):
   ;ITabularSequenceImpl::setToPrevious(ITabularSequenceImpl::Cursor&) const
   setToPrevious__20ITabularSequenceImplCFRQ2_20ITabularSequenceImpl6Cursor    @1034 NONAME
   ;ITabularSequenceImpl::allElementsDo(void*)
   allElementsDo__20ITabularSequenceImplFPv    @1035 NONAME
   ;ITabularSequenceImpl::removeAll(void*,void*)
   removeAll__20ITabularSequenceImplFPvT1    @1036 NONAME
   ;ITabularSequenceImpl::addAllFrom(const ITabularSequenceImpl&)
   addAllFrom__20ITabularSequenceImplFRC20ITabularSequenceImpl    @1037 NONAME
 ;From object file:  IIAVLKSS.C
  ;PUBDEFs (Symbols available from object file):
   ;IAvlKeySortedSetImpl::allElementsDo(void*,void*) const
   allElementsDo__20IAvlKeySortedSetImplCFPvT1    @1080 NONAME
   ;IAvlKeySortedSetImpl::isFirst
(const IAvlKeySortedSetImpl::Node*) const
   isFirst__20IAvlKeySortedSetImplCFPCQ2_20IAvlKeySortedSetImpl4Node    @1081 NONAME
   ;IAvlKeySortedSetImpl::setToPosition(unsigned long,IAvlKeySortedSetImpl::Node*&) const
   setToPosition__20IAvlKeySortedSetImplCFUlRPQ2_20IAvlKeySortedSetImpl4Node    @1082 NONAME
   ;IAvlKeySortedSetImpl::locateOrAddElementWithKey(const void*)
   locateOrAddElementWithKey__20IAvlKeySortedSetImplFPCv    @1083 NONAME

   ⋮
```

Data Mapping

The following section lists each data format and its equivalent C type in VisualAge C++ product, including the alignment and mapping for each.

Automatic Variables: When optimization is turned off (`/O-`), automatic variables have the same mapping as other variables, but they are mapped on the stack instead of in a data segment. Because memory on the stack is constantly reallocated on the stack, **automatic variables are not guaranteed to be retained after the return of the function that used them**.

When optimization is on, automatic variables are mapped as follows:

Size of Object	Alignment
1-byte	Byte-aligned
2-byte	Word-aligned
3-byte and greater	Doubleword-aligned

Note that the variables are ordered to minimize padding.

In VisualAge C++ product, a *word* consists of 2 bytes (or 16 bits) and a *doubleword* consists of 4 bytes (32 bits).

1. **Single-Byte Character**

Type	`signed char` and `unsigned char`
Alignment	Byte-aligned.
Storage mapping	Stored in 1 byte.

2. **Two-Byte Integer**

Type	`short` and its signed and unsigned counterparts
Alignment	Word-aligned.
Storage mapping	Byte-reversed, for example, `0x3B2C` (where `2C` is the least significant byte and `3B` is the most significant byte) is represented in storage as:

byte 0	byte 1
2C	3B

Toward high memory →

3. **Four-Byte Integer**

Type	`long, int,` and their signed and unsigned counterparts
Alignment	Doubleword-aligned.
Storage mapping	Byte-reversed, for example, `0x4A5D3B2C` (where `2C` is the least significant byte and `4A` is the most significant byte) is represented in storage as:

byte 0	byte 1	byte 2	byte 3
2C	3B	5D	4A

Toward high memory →

Note on IEEE Format:

In IEEE format, a floating point number is represented in terms of sign (`S`), exponent (`E`), and fraction (`F`):

$(-1)^{S} \times 2^{E} \times 1.F$

In the diagrams that follow, the first two rows number the bits. Read them vertically from top to bottom. The last row indicates the storage of the parts of the number.

4. **Four-Byte Floating Point (IEEE Format)**

Type	`float`
Alignment	Doubleword-aligned.
Bit mapping	In the internal representation, there is 1 bit for the sign (`S`), 8 bits for the exponent (`E`), and 23 bits for the fraction (`F`). The bits are mapped with the fraction in bit 0 to bit 22, the exponent in bit 23 to bit 30, and the sign in bit 31:

```
3 32222222 2221111111111
1 09876543 21098765432109876543210

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping — The storage mapping is as follows:

byte 0	**byte 1**	**byte 2**	**byte 3**
76543210	111111 54321098	22221111 32109876	33222222 10987654
FFFFFFFF	FFFFFFFF	EFFFFFFF	SEEEEEEE

Toward high memory →

5. **Eight-Byte Floating Point (IEEE Format)**

Type	`double`
Alignment	Doubleword-aligned on the 80386
Bit mapping	In the internal representation, there is 1 bit for the sign (S), 11 bits for the exponent (E), and 52 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 51, the exponent in bit 52 to bit 62, and the sign in bit 63:

```
6 66655555555 5544444444443333333333222222222211111111111
3 21098765432 1098765432109876543210987654321098765432109876543210

S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping — The storage mapping is as follows:

byte 0	**byte 1**	**byte 2**	**...**
76543210	111111 54321098	22221111 32109876	...
FFFFFFFF	FFFFFFFF	FFFFFFFF	...

Toward high memory →

byte 5	**byte 6**	**byte 7**
44444444 76543210	55555544 54321098	66665555 32109876
FFFFFFFF	EEEEFFFF	SEEEEEEE

Toward high memory →

6. **Ten-Byte Floating Point in Sixteen-Byte Field (IEEE Format)**

Type	`long double`
Alignment	Doubleword-aligned on the 80386
Bit mapping	In the internal representation, there is 1 bit for the sign (`S`), 15 bits for the exponent (`E`), and 64 bits for the fraction (`F`). The bits are mapped with the fraction in bit 0 to bit 63, the exponent in bit 64 to bit 78, and the sign in bit 79:

```
7 777777777666666
9 876543210987654

S EEEEEEEEEEEEEEE

6666555555555544444444443333333333222222222211111111111
3210987654321098765432109876543210987654321098765432109876543210

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping — The storage mapping is as follows:

byte 0	**byte 1**	**byte 2**	**...**
76543210	111111 54321098	22221111 32109876	...
FFFFFFFF	FFFFFFFF	FFFFFFFF	...

Toward high memory →

byte 7	**byte 8**	**byte 9**
66666555 43210987	77666666 10987654	77777777 98765432
FFFFFFFF	EEEEEEEE	SEEEEEEE

Toward high memory →

7. **Null-Terminated Character Strings**

Type	`char string[n]`
Size	Length of string (not including null).
Alignment	Byte-aligned. If the length of the string is greater than a doubleword, the string is doubleword-aligned.
Storage mapping	The string `"STRING"` is stored in adjacent bytes as:

byte 0	**byte 1**	**byte 2**	**byte 3**	**byte 4**	**byte 5**	**byte 6**
`'S'`	`'T'`	`'R'`	`'I'`	`'N'`	`'G'`	`'\0'`

Toward high memory →

8. **Fixed-Length Arrays Containing Simple Data Types**

Type
: The corresponding VisualAge C++ declaration depends on the simple data type in the array. For an array of `int`, for example, you would use something like:

```
int int_array[n];
```

For an array of `float`, you would use something like:

```
float float_array[n];
```

Size
: `n * (s + p)`, where *n* is the number of elements in the array, *s* is the size of each element, and *p* is the alignment padding.

Alignment
: The alignment is the same as that of the simple data type of the array elements. For instance, an array of `short` elements would be word-aligned, while an array of `int` elements would be doubleword-aligned. If the length of the array is greater than a doubleword, the array is doubleword-aligned.

Storage mapping
: The first element of the array is placed in the first storage position. For multidimensional arrays, row-major ordering is used.

9. **Aligned Structures**

Type	`struct`
Size	Sum of the sizes for each type in the `struct` plus padding for alignment.
Alignment	The first element of the structure is aligned according to the alignment rule of the element that has the most restrictive alignment rule. If the length of the structure is greater than a doubleword, the structure is doubleword-aligned. The alignment of the individual members is not changed. In the following example, types `char`, `short`, and `float` are used in the struct. Because `float` must be aligned on the doubleword boundary, and because this is the most restrictive alignment rule, the first element must be aligned on the doubleword boundary even though it is only a `char`.

Note: The first element will not necessarily occupy a doubleword, but it will be aligned on it.

```
struct y {
          char char1;     /* aligns on doubleword */
          short short1;   /* aligns on word */
          char char2;     /* aligns on byte */
          float float1;   /* aligns on doubleword */
          char char3      /*aligns on byte */
        };
```

Storage mapping — The `struct` is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
`char1`	pad	`short1`	`short1`	`char2`	pad

Toward high memory →

byte 6	byte 7	byte 8	byte 9	byte 10
pad	pad	`float1`	`float1`	`float1`

Toward high memory →

byte 11	byte 12	byte 13	byte 14	byte 15
`float1`	`char3`	pad	pad	pad

Toward high memory →

Note: This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

10. **Unaligned or Packed Structures**

Type

The definition of the structure variable is preceded by the keyword `_Packed`, or the **`#pragma pack`** directive or `/Sp` option is used. For instance, the following definition would create a packed `struct` called `mystruct` with the type `struct y` (defined above):

```
_Packed struct y mystruct
```

Size

The sum of the sizes of each type that makes up the `struct`.

Storage mapping

When the `_Packed` keyword, the **`#pragma pack(1)`** directive, or `/Sp(1)` option is used, the structure `mystruct` is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4
char1	short1	short1	char2	float1

Toward high memory →

byte 5	byte 6	byte 7	byte 8
float1	float1	float1	char3

Toward high memory →

When **`#pragma pack(2)`** or the `/Sp(2)` option is used, `mystruct` is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
char1	pad	short1	short1	char2	pad

Toward high memory →

byte 6	byte 7	byte 8	byte 9	byte 10	byte 11
float1	float1	float1	float1	char3	pad

Toward high memory →

Note: This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

11. **Arrays of Structures**

Type	The definition for an array of `struct` would look like: `struct y mystruct_array[n]` The definition of an array of `_Packed struct` would look like: `_Packed struct y mystruct_array[n]`
Alignment	Each structure is aligned according to the structure alignment rules. This may cause a fixed-length gap between consecutive structures. In the case of packed structures, there is no padding.
Storage mapping	The first element of the array is placed in the first storage position. Row-major ordering is used for multidimensional arrays. **Note:** This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

12. **Structures Containing Bit Fields**

Type	`struct`
Size	The sum of the sizes for each type in the `struct` plus padding for alignment.
Alignment	Each structure is aligned according to the structure alignment rules.

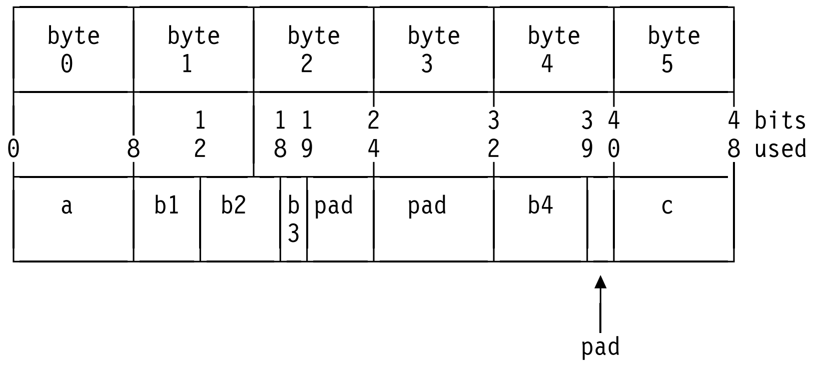
Data Mapping

Storage mapping

Given the following structure:

```
struct s {
        char a;
        int b1:4;
        int b2:6;
        int b3:1;
        int   :0;
        int b4:7;
        char c;
      }
```

`struct s` would be stored as follows:



Notes:

a. The second row of the table counts the number of bits used and should be read vertically top-to-bottom. Bits are allocated from least-significant to most-significant within each byte. In the diagram above, the least-significant bit is on the left.

b. This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

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

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot have a direct object of an abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

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 specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

additional heap. (1) A *Language Environment* heap created and controlled by a call to CEECRHP. See also *below heap, anywhere heap,* and *initial heap*.

address space. (1) The range of addresses available to a computer program. *ANSI.* (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) In the AIX operating system, the code, stack, and data that are accessible by a process. (4) The area of virtual storage available for a particular job. (5) The memory locations that can be referenced by a process. *X/Open. ISO.1.*

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) An array or a class object with no private or protected members, no constructors, no base classes, and no virtual functions. (4) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1.*

alert. (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM*. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

American National Standards Institute. See *ANSI*.

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase "enclosed in angle 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 <less-than-sign> and <greater-than-sign> are used. *X/Open*.

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*.

anywhere heap. The VisualAge C++ heap controlled by the ANYHEAP run-time option. It contains library data, such as VisualAge C++ control blocks and data structures not normally accessible from user code. The anywhere heap may reside above 16M. See also *below heap, additional heap, initial heap*.

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 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).

assembler user exit. In the *Language Environment* a routine to tailor the characteristics of an enclave prior to its establishment.

automatic data. Data that does not persist after a routine has finished executing. Automatic data may be automatically initialized to a certain value upon entry and reentry to a routine.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

backslash. The character \. This character is named <backslash> in the portable character set.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

based on. The use of existing classes for implementing new classes.

below heap. The VisualAge C++ heap controlled by the BELOWHEAP runtime option, which contains library data, such as VisualAge C++ control block and data structures not normally accessible from user code. Below heap always resides below 16M. See also *anywhere heap, initial heap, additional heap*.

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*.

bit field. A member of a structure or union that contains a specified number of bits. *IBM*.

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.

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*.

breakpoint. A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. *ISO Draft*.

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*.

caller. A routine that calls another routine.

carriage-return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. *X/Open*.

CASE (Computer-Aided Software Engineering). A set of tools or programs to help develop complex applications. *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.

class name. A unique identifier of a class type that becomes a reserved word within its scope.

class template. A blueprint describing how a set of related classes can be constructed.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

client program. A program that uses a class. The program is said to be a *client* of the class.

COBOL (Common Business-Oriented Language). A high-level language, based on English, that is primarily used for business applications.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI*.

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.

code point. (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

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.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open*.

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.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the *Language Environment* and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

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*.

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

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.

coordinated universal time (UTC). Equivalent to Greenwich Mean Time (GMT)

copy constructor. A constructor that copies a class object of the same class type.

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) In DOS, the directory that is searched when a file name is entered with no indication of the directory that lists the file name. DOS assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (4) In the AIX operating system, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

cursor. A reference to an element at a specific position in a data structure.

D

data definition (DD). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM*. (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI*. (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname*.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data set. Under MVS, a named collection of related data records that is stored and retrieved by an assigned name. Equivalent to a CMS *file*.

data structure. The internal data representation of an implementation.

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*.

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In MVS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *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 constructor. A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

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`.

demangling. The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

denormal. Pertaining to a number with a value so close to `0` that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. *PL/I* control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one *PL/I* routine to another during run time.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

device. A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

difference. Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B. 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, if *m*>*n*, the difference contains that element *m-n* times. If *m*≤*n*, the difference contains that element zero times.

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.*

doubleword. A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. *IBM.*

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.*

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic storage. Synonym for *automatic storage*.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters. *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.*

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In the Language Environment for MVS and VM, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

enumeration constant. In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM*.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM*.

equivalence class. (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM*. (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open*.

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, nonprinting characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at runtime can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM*.

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*.

executable program. A program that has been link-edited and therefore can be run in a processor. *IBM*.

extension. (1) An element or function not included in the standard language. (2) File name extension.

F

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.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

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*.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a **;** (semicolon). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

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.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

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*.

include statement. In the C and C++ languages, a preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file. *IBM*.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures and unions of unspecified content. A void type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

indirection. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to.

inheritance. A technique that allows the use of an existing class as the base for creating other classes.

initial heap. The VisualAge C++ heap controlled by the HEAP runtime option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. In the C++ language, there are three types of initializers:

1. An expression followed by an assignment operator is used to initialize fundamental data type objects or class objects that have copy constructors.
2. An expression enclosed in braces ({ }) is used to initialize aggregates.
3. A parenthesized expression list is used to initialize base classes and members using constructors.

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;
```

instantiate. To create or generate a particular instance or object of a data type. For example, an instance `box1` of class `box` could be instantiated with the declaration:

```
box box1;
```

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.

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

integer constant. A decimal, octal, or hexadecimal constant.

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*.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

K

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*.

Language Environment. Abbreviated form of IBM Language Environment for MVS and VM. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

last element. The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

lexically. Relating to the left-to-right order of units.

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.

line. A sequence of zero or more non-new-line characters plus a terminating new-line character. *X/Open*.

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*.

loader. A routine, commonly a computer program, that reads data into main storage. *ANSI*.

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft*.

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*.

localization. The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open*.

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.

main function. An external function with the identifier `main` that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named `main`.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. The prelinker uses these mangled names to ensure type-safe linkage. See also *demangling*.

map file. A listing file that can be created during the prelink or link step and that contains information on the size and mapping of segments and symbols.

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.

member function. (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

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.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open.*

multiple inheritance. An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

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.

nested class. A class defined within the scope of another class.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open.*

node. In a tree structure, a point at which subordinate items of data originate. *ANSI.*

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*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. (1) A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

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*.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

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.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM*.

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.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

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.

pipe. To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|). Two commands connected in this way constitute a pipeline. *IBM*.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

pointer to member. An operator used to access the address of non-static members of a class.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

`<NUL>`	
`<alert>`	
`<backspace>`	
`<tab>`	
`<newline>`	
`<vertical-tab>`	
`<form-feed>`	
`<carriage-return>`	
`<space>`	
`<exclamation-mark>`	`!`
`<quotation-mark>`	`"`
`<number-sign>`	`#`
`<dollar-sign>`	`$`
`<percent-sign>`	`%`
`<ampersand>`	`&`
`<apostrophe>`	`'`
`<left-parenthesis>`	`(`
`<right-parenthesis>`	`)`
`<asterisk>`	`*`
`<plus-sign>`	`+`
`<comma>`	`,`
`<hyphen>`	`-`
`<hyphen-minus>`	`-`
`<period>`	`.`
`<slash>`	`/`
`<zero>`	`0`
`<one>`	`1`
`<two>`	`2`
`<three>`	`3`
`<four>`	`4`
`<five>`	`5`
`<six>`	`6`
`<seven>`	`7`
`<eight>`	`8`
`<nine>`	`9`
`<colon>`	`:`
`<semicolon>`	`;`
`<less-than-sign>`	`<`
`<equals-sign>`	`=`
`<greater-than-sign>`	`>`
`<question-mark>`	`?`
`<commercial-at>`	`@`

<A>	A
<B>	B
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I
<J>	J
<K>	K
<L>	L
<M>	M
<N>	N
<O>	O
<P>	P
<Q>	Q
<R>	R
<S>	S
<T>	T
<U>	U
<V>	V
<W>	W
<X>	X
<Y>	Y
<Z>	Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>	]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	_
<grave-accent>	`
<a>	a
<b>	b
<c>	c
<d>	d
<e>	e
<f>	f
<g>	g
<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m
<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	|
<right-brace>	}
<right-curly-bracket>	}
<tilde>	˜

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

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.

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

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. A process is created by another process issuing the `fork`[] function. The process that issues the `fork`[] function is known as the parent process, and the new process created by the `fork`[] function is known as the child process. *X/Open. ISO.1.*

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

Q

qualified name. Used to qualify a nonclass type name such as a member by its class name.

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

R

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

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.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open. ISO.1.*

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

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

scalar. An arithmetic object, or a pointer to an object of any type.

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. An object used by multithread applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership. There can be multiple process groups in the same session. *X/Open. ISO.1.*

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open.*

This feature is provided as part of OpenEdition MVS Shell and Utilities feature licensed program.

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.* (3) In AIX operating system operations, a method of interprocess communication that simulates software interrupts. *IBM.*

signal handler. A function to be called when the signal is reported.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

S-name. An external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

source file. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM*.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM*.

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.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input. (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. (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.

storage class specifier. One of: auto, register, static, or extern.

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*.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

string literal. 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.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile. *IBM*.

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1*. *ANSI*. (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions with variable types.

template class. A class instance generated by a class template.

template function. A function generated by a function template.

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. The smallest unit of operation to be performed within a process. *IBM*.

tilde. The character ˜. This character is named <tilde> in the portable character set.

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 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

undefined behavior. Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

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.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

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*.

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

`<left-square-bracket>`	[
`<right-square-bracket>`	]
`<left-brace>`	{
`<right-brace>`	}
`<backslash>`	\
`<circumflex>`	^
`<tilde>`	~
`<exclamation-mark>`	!
`<number-sign>`	#
`<vertical-line>`	|
`<grave-accent>`	`
`<dollar-sign>`	$
`<commercial-at>`	@

virtual function. A function of a class that is declared with the keyword `virtual`. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at run time.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

W

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*.

word boundary. Any storage position at which data must be aligned for certain processing operations. The halfword boundary must be divisible by 2; the fullword boundary by 4; and the doubleword boundary by 8. *IBM*.

working directory. Synonym for *current working directory*.

write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI*.

Bibliography

This bibliography lists the publications that make up the IBM VisualAge C++ library and publications of related IBM products referenced in this book. The list of related publications is not exhaustive but should be adequate for most VisualAge C++ users.

The IBM VisualAge C++ Library

The following books are part of the IBM VisualAge C++ library.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building VisualAge C++ Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963-00
- *C Library Reference*, S25H-6964

The IBM VisualAge C++ BookManager Library

The following documents are available in VisualAge C++ in BookManager format.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building VisualAge C++ Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963-00
- *C Library Reference*, S25H-6964

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])*
- *Draft Proposed American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)*

IBM OS/2 2.1 Publications

The following books describe the OS/2 2.1 operating system and the Developer's Toolkit 2.1.

- *OS/2 2.1 Using the Operating System*, S61G-0703
- *OS/2 2.1 Installation Guide*, S61G-0704
- *OS/2 2.1 Quick Reference*, S61G-0713
- *OS/2 2.1 Command Reference*, S71G-4112
- *OS/2 2.1 Information and Planning Guide*, S61G-0913
- *OS/2 2.1 Keyboard and Codepages*, S71G-4113
- *OS/2 2.1 Bidirectional Support*, S71G-4114
- *OS/2 2.1 Book Catalog*, S61G-0706
- *Developer's Toolkit for OS/2 2.1: Getting Started*, S61G-1634

IBM OS/2 3.0 Publications

- *User's Guide to OS/2 Warp*, G25H-7196-01

The following books make up the OS/2 3.0 Technical Library (G25H-7116).

- *Control Program Programming Guide*, G25H-7101
- *Control Program Programming Reference*, G25H-7102

- *Presentation Manager Programming Guide - The Basics*, G25H-7103
- *Presentation Manager Programming Guide - Advanced Topics*, G25H-7104
- *Presentation Manager Programming Reference*, G25H-7105
- *Graphics Programming Interface Programming Guide*, G25H-7106
- *Graphics Programming Interface Programming Reference*, G25H-7107
- *Workplace Shell Programming Guide*, G25H-7108
- *Workplace Shell Programming Reference*, G25H-7109
- *Information Presentation Facility Programming Guide*, G25H-7110
- *OS/2 Tools Reference*, G25H-7111
- *Multimedia Application Programming Guide*, G25H-7112
- *Multimedia Subsystem Programming Guide*, G25H-7113
- *Multimedia Programming Reference*, G25H-7114
- *REXX User's Guide*, S10G-6269
- *REXX Reference*, S10G-6268

Other Books You Might Need

The following list contains the titles of IBM books that you might find helpful. These books are not part of the VisualAge C++ or OS/2 libraries.

BookManager READ/2 Publications

- *IBM BookManager READ/2: General Information*, GB35-0800
- *IBM BookManager READ/2: Getting Started and Quick Reference*, SX76-0146
- *IBM BookManager READ/2: Displaying Online Books*, SB35-0801
- *IBM BookManager READ/2: Installation*, GX76-0147

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.
- *OS/2 C++ Class Library: Power GUI Programming with C Set* ++ by Kevin Leong, William Law, Robert Love, Hiroshi Tsuji, and Bruce Olson, John Wiley & Sons, Inc.

Index

Communicating Your Comments to IBM

IBM VisualAge C++ for OS/2
Programming Guide

Version 3.0

Publication No. S25H-6958-00

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 C++ for OS/2
Programming Guide

Version 3.0

Publication No. S25H-6958-00

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
S25H-6958-00

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

S25H-6958-00

Cut or Fold
Along Line

Part Number: 25H6958
Program Number: 30H1664
30H1665
30H1666

Printed in U.S.A.

25H6958

S25H-6958-00