



Dev-C++ is a full-featured *Integrated Development Environment* (IDE) for the C/C++ programming language. It uses [Mingw](#) port of GCC (GNU Compiler Collection) as its compiler. It can create native Win32 executables, either console or GUI, as well as DLLs and static libraries. Dev-C++ can also be used in combination with Cygwin or any other GCC based compiler.

Dev-C++ is a [Free Software](#) distributed under the terms of the [GNU General Public License \(GPL\)](#)

Dev-C++ features are :

- Support GCC based compilers (Mingw included)
- Integrated debugging (with GDB)
- Support for multiple languages (localization)
- Class Browser
- Debug variable Browser
- Code Completion
- Function Listing
- Project Manager
- Customizable syntax highlighting editor
- Quickly create Windows, console, static libraries and DLL
- Support of templates for creating your own project types
- Makefile creation
- Edit and compile Resource files
- Tool Manager
- Print support
- Find and replace facilities
- Package manager, for easy installation of add-on libraries

License Agreement

Bloodshed Dev-C++ is distributed under the GNU General Public License. Be sure to read it before using Dev-C++.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by

the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the

source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we

want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program"

means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of

running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the

notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1

above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively

when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but

does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based

on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the

entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program

with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections

1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer

to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to

control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent

access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under

this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the

Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of

this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS

TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED

TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it

free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>  
Copyright (C) 19yy <name of author>
```

```
This program is free software; you can redistribute it and/or modify  
it under the terms of the GNU General Public License as published by  
the Free Software Foundation; either version 2 of the License, or  
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,  
but WITHOUT ANY WARRANTY; without even the implied warranty of  
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License  
along with this program; if not, write to the Free Software  
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this

when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.

This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License. Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs. If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library. If this is what you want to do, use the GNU Library General
Public License instead of this License.

System Requirements

These are the minimum requirements of Dev-C++:

- Microsoft Windows 95, 98, NT 4, 2000, XP
- 8 MB RAM with a big swapfile
- 100 Mhz Intel compatible CPU
- 30 MB free disk space

These are the recommended requirements of Dev-C++:

- Microsoft Windows 2000, XP
- 32 MB RAM
- 400 Mhz Intel compatible CPU
- 200 MB free disk space

Credits

Developers : Colin Laplace, Hongli Lai, Mike Berg, Yiannis Mandravellos

Mingw compiler system :Mumit Khan, J.J. Var Der Heidjen, Colin Hendrix and GNU developers

Update system and initial work on the help file : Kip Warner

New Look theme :Gerard Caulfield:

Gnome icons :Gnome designers

Blue theme :Thomas Thron

Introduction to C :University of Leicester

Creating a Project

What is a Dev-C++ Project ?

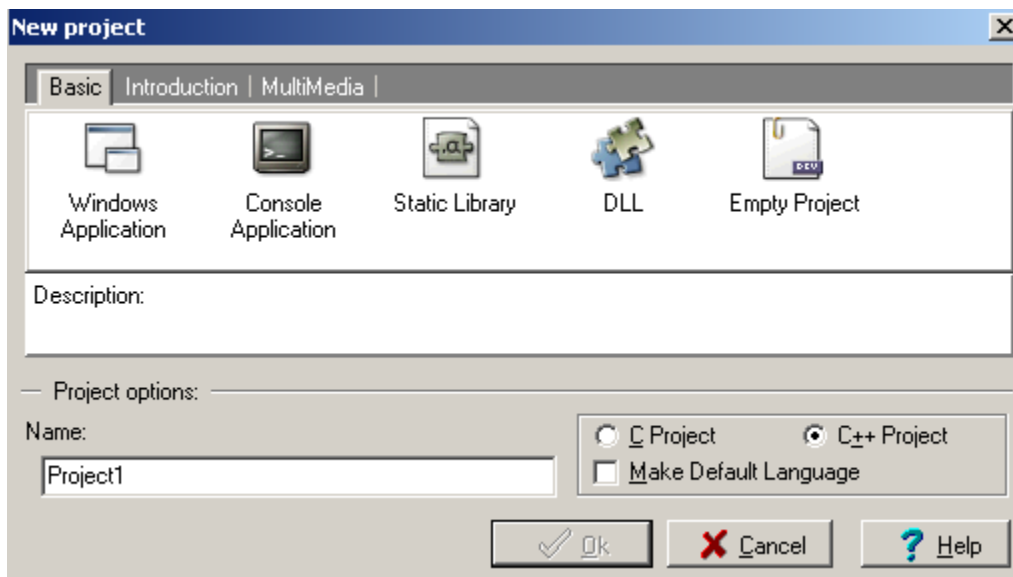
A Project is a center for managing your different source files and options inside Dev-C++. It helps you navigate through your code, and easily set different parameters, like the type of program you are doing (GUI, console, DLL ...).

When to use Dev-C++ Projects ?

- If you have more than one source file, you must create a project in order for Dev-C++ to link all your source files together, after they were compiled.
- If you need to create a DLL or static library, or want to use Resources files in your program

How can i create a Dev-C++ Project ?

Go to the *File* menu and click on *New*, then *Project*. A dialog opens, containing different Project types.



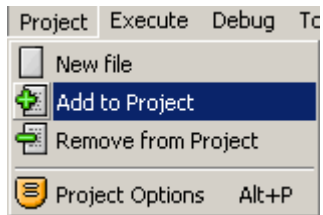
Here are the different basic projects type :

- Windows application : creates a Windows program, using the Win32 API.
- Console application : creates a console program
- Static library : creates an empty project with the options needed for building a static library
- DLL : creates a Win32 Dynamic Link Library

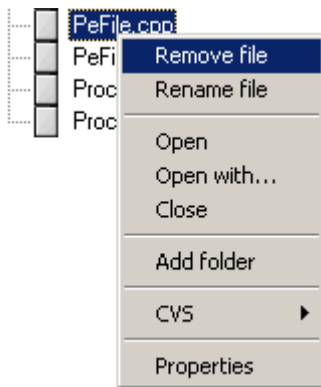
Now that you selected your project type, fill the name of your project in the corresponding edit box, select the language you will be using (C or C++), and click on OK.

Adding and Removing files

Adding and removing files is very easy. You can add multiple files in the same time by clicking on *Project* menu, then on *Add to Project*.



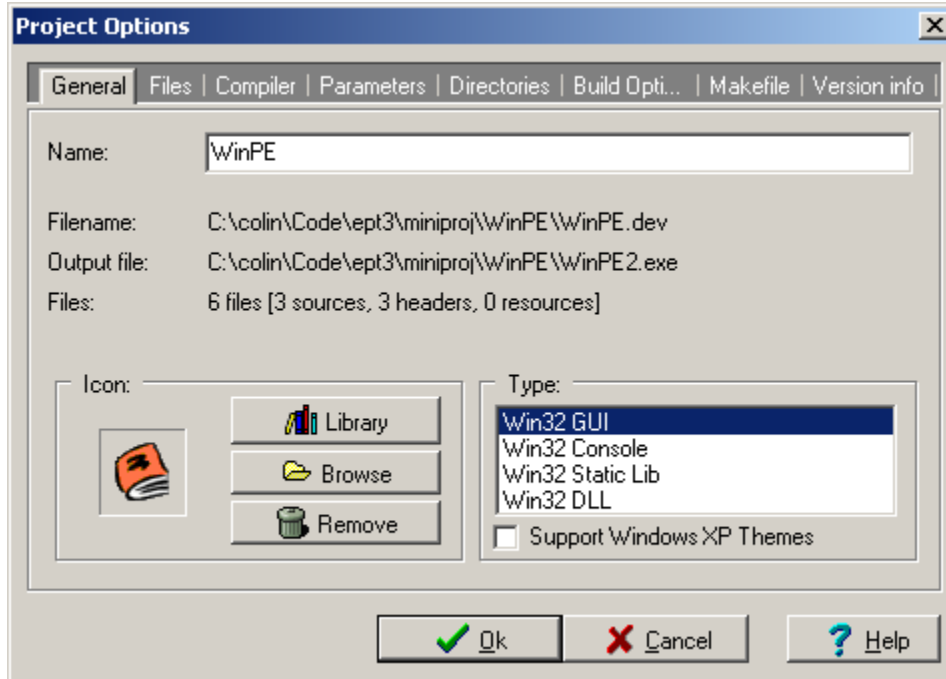
You have different ways for removing files from your project. Either click on *Project* menu, then on *Remove from Project* and select the file you want to remove in the list, or right-click on the file you want to remove in the Project Manager, and click on *Remove file*



Project options overview

You can load the Project Options dialog by clicking on the *Project* menu, then on *Options* (shortcut :Alt+P).

General sheet :



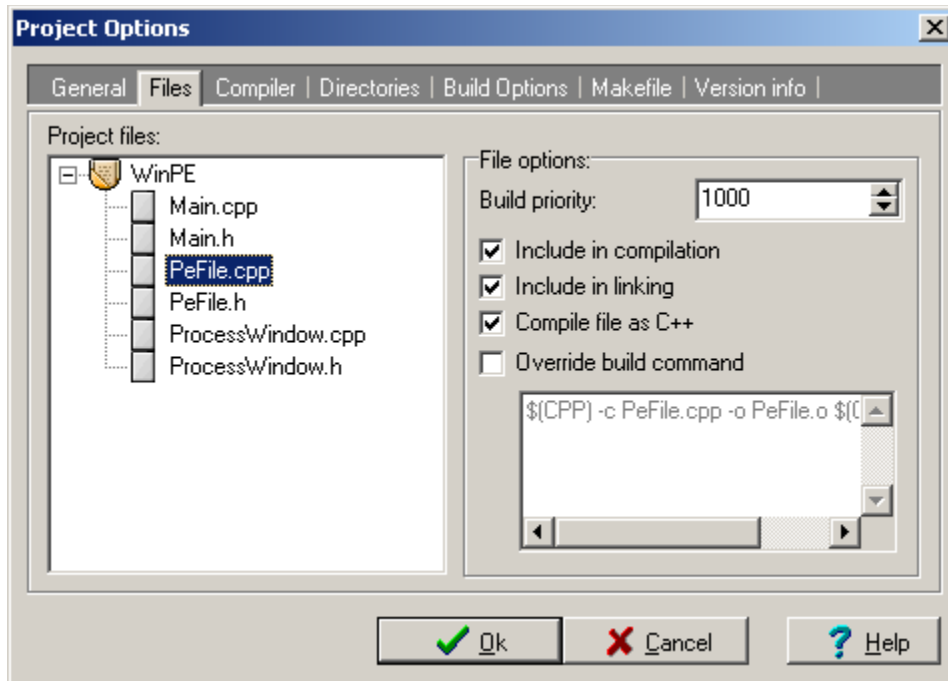
Name: Modify here the name of your project.

Icon : You can assign an icon to your program, either by selecting one in the Icon Library, or by giving your own icon using the Browse button.

Type : This is an important settings which indicates which project type you are making. Select :

- **Win32 GUI:** if your application is a graphical Windows program
- **Win32 Console :** if your application needs a console window (MS-Dos window)
- **Win32 Static Lib :** if you are creating a static library
- **Win32 DLL :** if you are creating a dynamic link library (DLL)

Files sheet :



This window enables you to modify the compilation commands and options for each source file.

Build priority : Increment this value in order to have the source file compiled in priority of the others

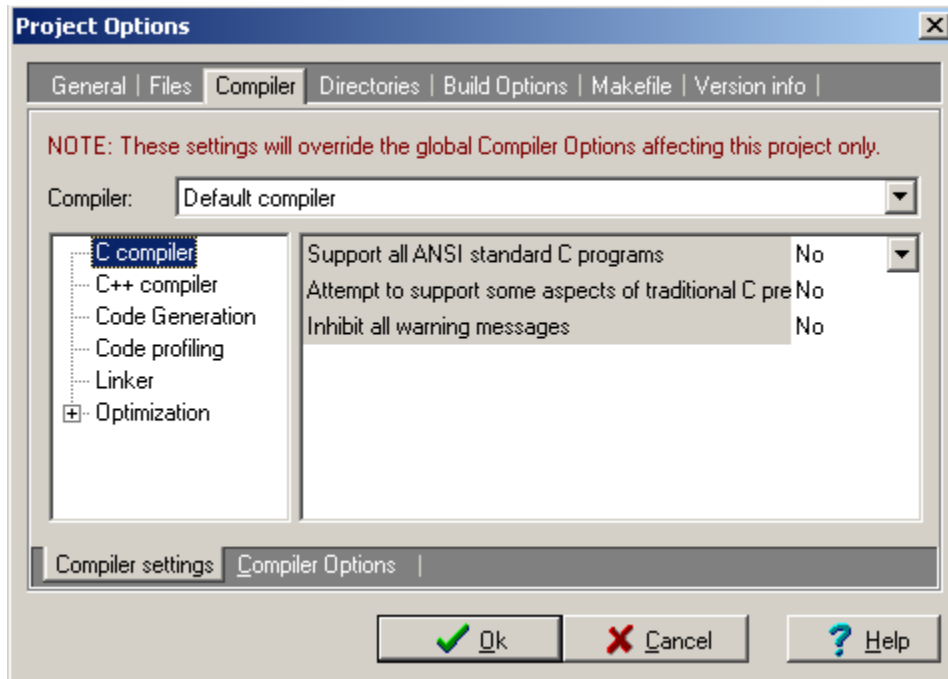
Include in compilation : If not set, your file will not be compiled

Include in linking : Add the object file generated from the source file to the linking stage

Compile file as C++ : Check this flag if it is a C++ source file

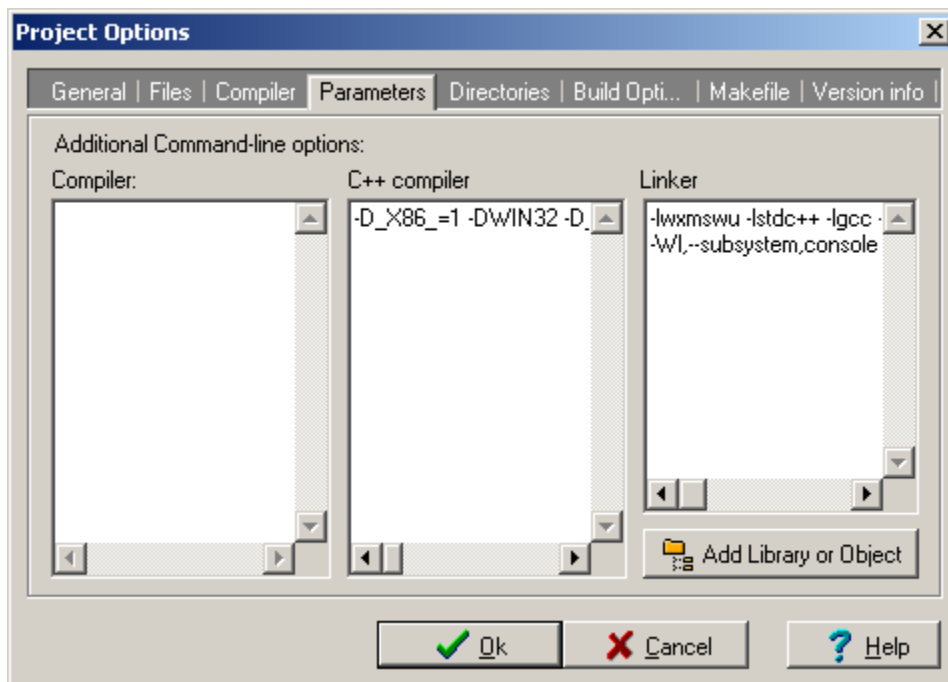
Override build command : For experimented users only. You can change there the command used by Dev-C++ to compile your file

Compiler sheet :



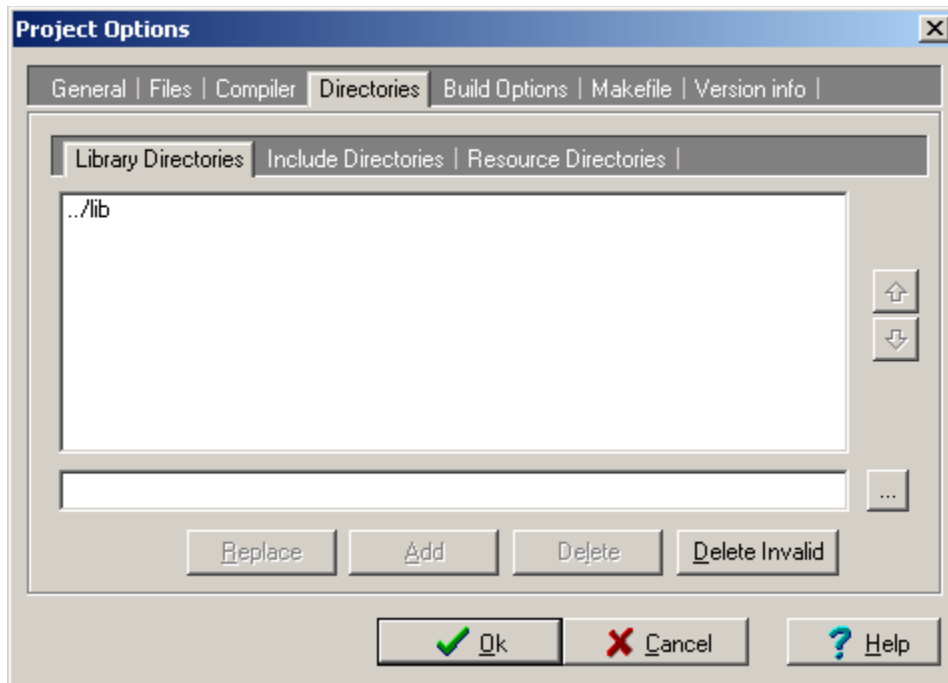
See [Compiler Options](#)ID_COMPILEROPTIONS

Parameters sheet :



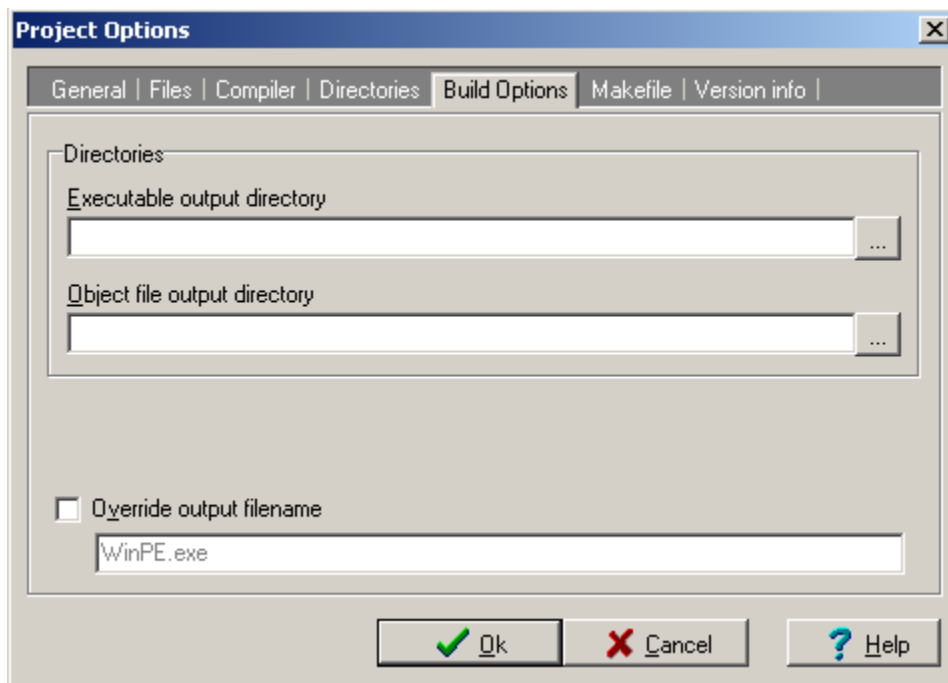
You can provide here command line arguments to the C/C++ compilers and linker. Use the linker parameters box to specify libraries to link with your project. For more information, please [read this](#)ID_LINKLIB.

Directories sheet :



You can provide here a list of Includes, Resources and Libraries directories to be searched when compiling/linking.

Build Options sheet :

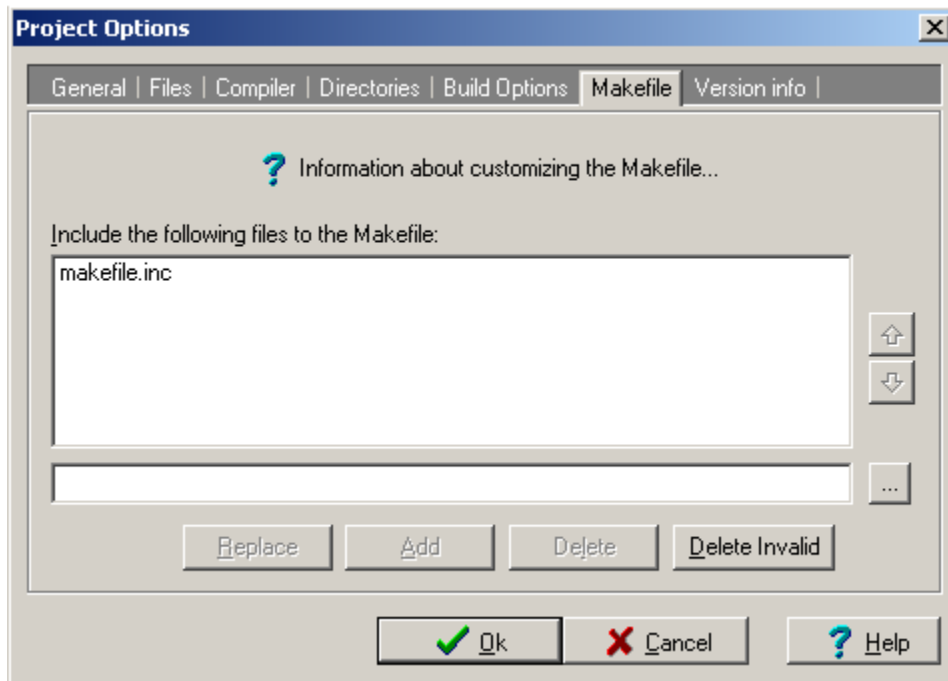


Executable output directory : Specify here the directory where your executable will be created (default is project's directory).

Object file output directory : Specify here the directory where your object files will be created (default is source file's directory).

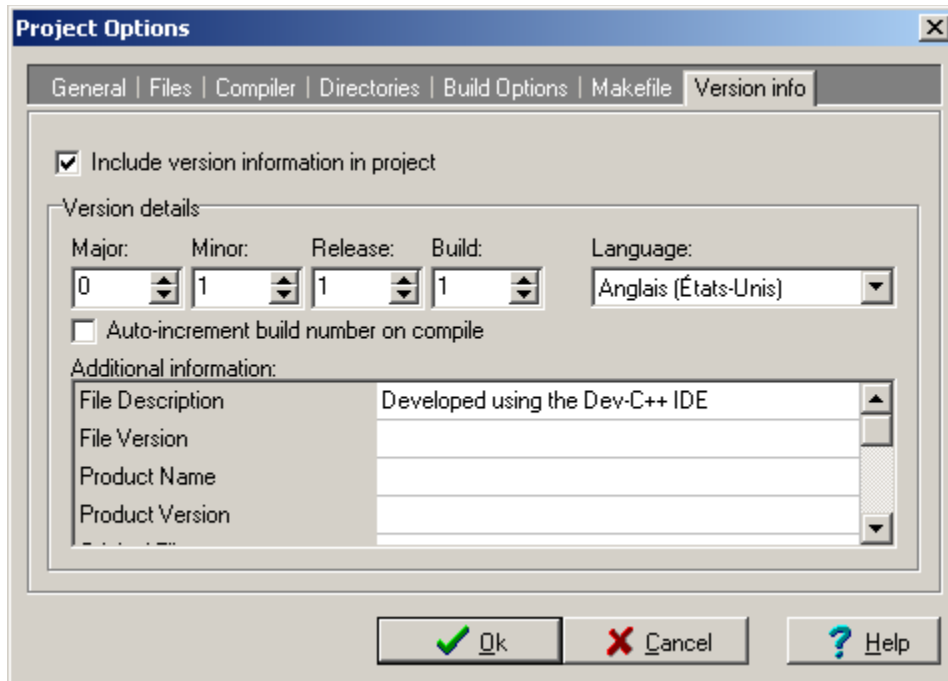
Override output filename : You may change the output filename of your program here.

Makefile sheet :



Dev-C++ automatically creates a Makefile for taking care of the building process. If you are experienced with makefiles and want to add other lines, you can do it here.

Version Info sheet :



You can specify version information for your program here.

Linking libraries with your project

A little history :

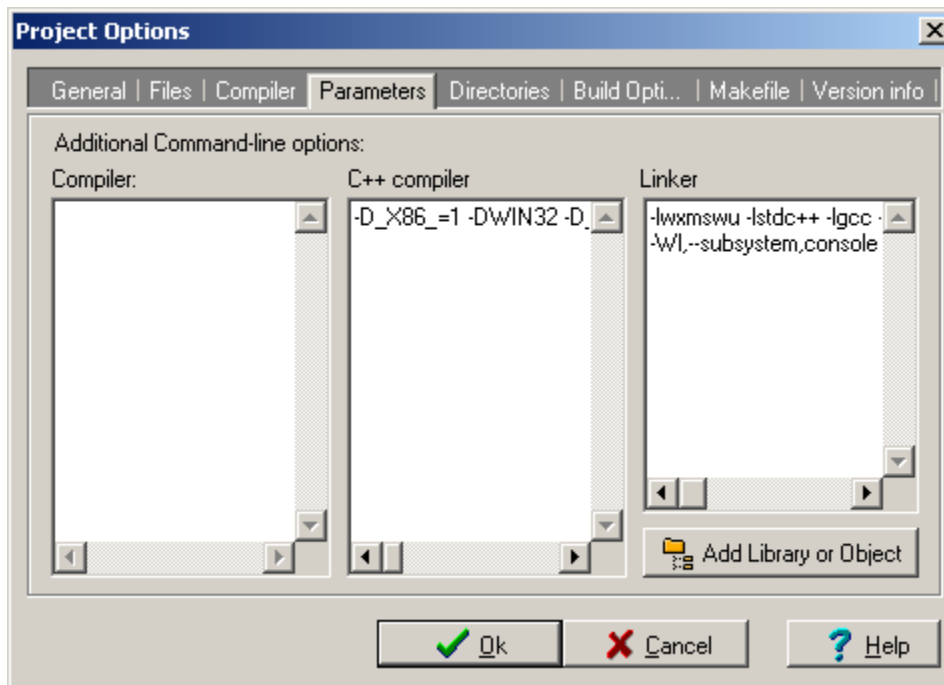
Library filenames under the GNU system are in the form *libNAME.a* (with NAME the name of the library, like wsock32).

Thus if for example you want to use the wsock32 (winsock) library, the filename will be *libwsock32.a*

The GCC parameter for linking a library is **-NAME**, so for linking with the wsock32 library we would give GCC the -lwsock32 parameter.


Linking your library:

Click on *Project* menu then on *Options*. Now click on the *Parameters* sheet.



In the *Linker* edit box, you can specify as many libraries as you need. You can also pass the complete filename of the library.

Example : `-lwx -lm -lwsock32 c:\libs\mylib.a c:\objs\myobj.o`

You may also use the *Add Library or Object* button  to select your library from a list.


Compiling and linking process

How does the compile and link process works ?

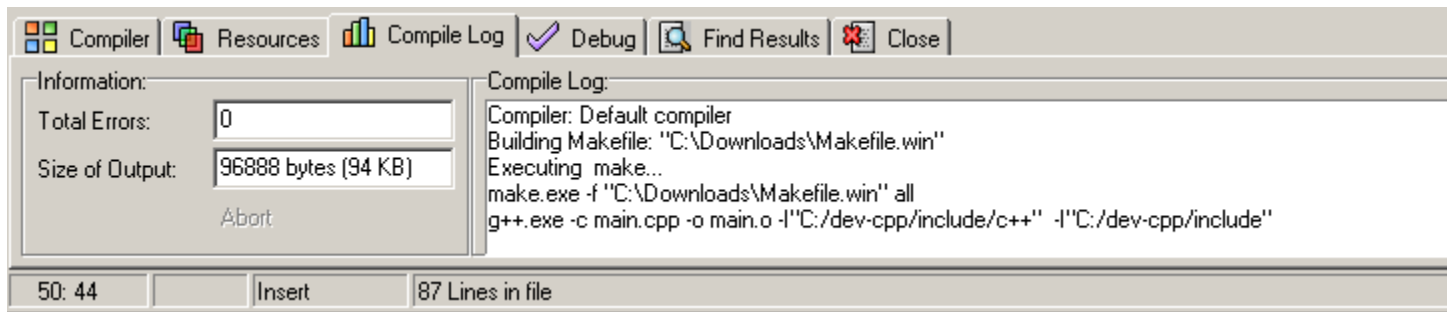
The process can be globally divided into several programs :

- **Preprocessor**: process the source file before actually compiling it. It will for example check and replace macros and include files in your source.
- **Compiler** : transform your source file into assembly code (processor language – human readable)
- **Assembler** :takes the assembly code and generate machine-readable code (binary object code)
- **Linker** : assembles and resolves object codes together to create a single executable.

How to do this in Dev-C++ ?

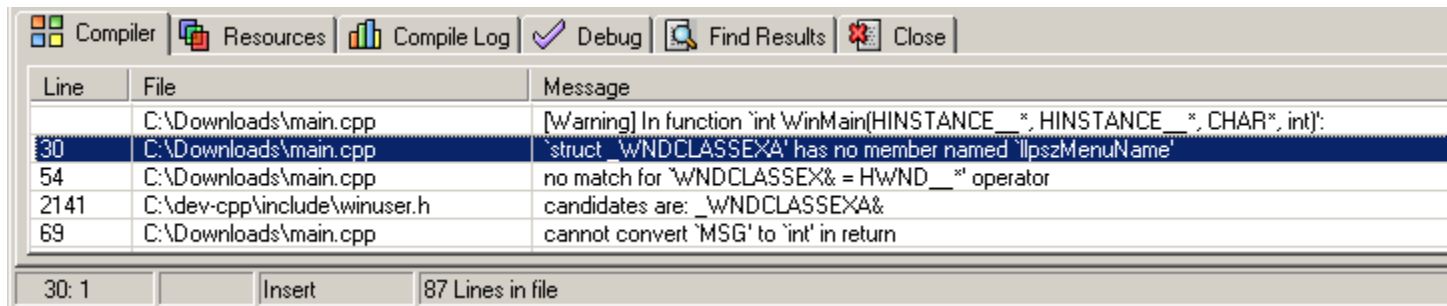
Just go to the *Execute* menu and click *Compile*  (shortcut : Ctrl+F9), and Dev-C++ will take care of calling the compiler and linker.

Look at the bottom panel of Dev-C++, you should get something like this :



This log window shows you what Dev-C++ is doing (the program it executes, the files it is creating, ...). In case your program compiled, you will see the message 'Compilation successful' in this window.


If compiling or linking fails, you will see a list of the errors on the bottom panel :




You can directly jump to the line in your code where the error appear by double-clicking on its item. To note: a 'parse error' is a syntax error (like a missing semicolon at the end of an instruction, etc...).

Executing your program

Basics :

Executing your program is as simple as clicking on the *Execute* menu, then *Run*  (shortcut : Ctrl+F10).

You can also use *Compile and Run*  (shortcut : F9) to build your program before executing it.

If you need to pass parameters to your program, you can use *Execute* menu, then *Parameters* to provide them.

Profiling :

This process reports you what happens in your program (function calls, execution time, ...) when you execute it. You must enable 'Generate profiling info for analysis' in Compiler Options, Code Generation sheet. Then compile your project as usual, and run it (profiling information are generated/refreshed each time you execute your program).

When you are done, click the *Execute* menu, then *Profile analysis*. The profiling window will appear and show you a profile of the execution of your program.


Introduction to debugging

It is a known fact that sometimes developers make errors when making a program. Those errors commonly called 'bugs' can be very tricky to find and correct, thus the debugger has been created to help developers investigate their program while it is running.

A debugger basically runs a program while keeping track of its functions, variables and instructions. It is capable of stopping your program at a given moment, which is called breakpointing. You can set breakpoints anywhere in your code : once your program reaches that code during execution, the debugger will stop and let you examine the current data in your program.

You probably already understand how useful a debugger may be when trying to correct bugs, so you may go to the [next section](#) ID_DEBUGPROCESS to learn how to use Dev-C++ integrated debugging.

Debugging your program

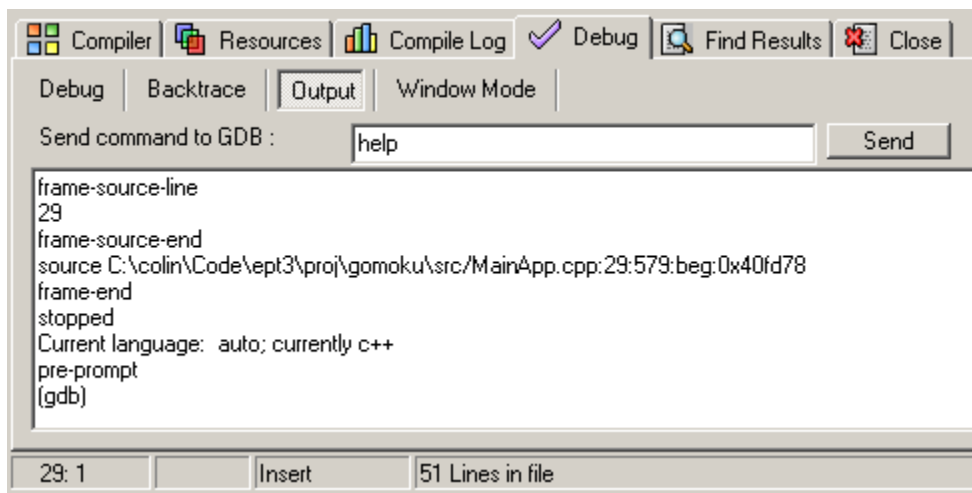
Launching your program into the debugger is easy. Just go to the *Debug* menu and click on *Debug*  (shortcut : F8).

If you do not have debugging information set in your project, Dev-C++ will ask you if you want to rebuild your program with these information enabled. You can manually select that option in Compiler Option in the Linker section. After your project has been rebuild, you can launch Debug again.

The debugger has now loaded your program and runs it. but until that you can learn to control several aspects of the debugger :

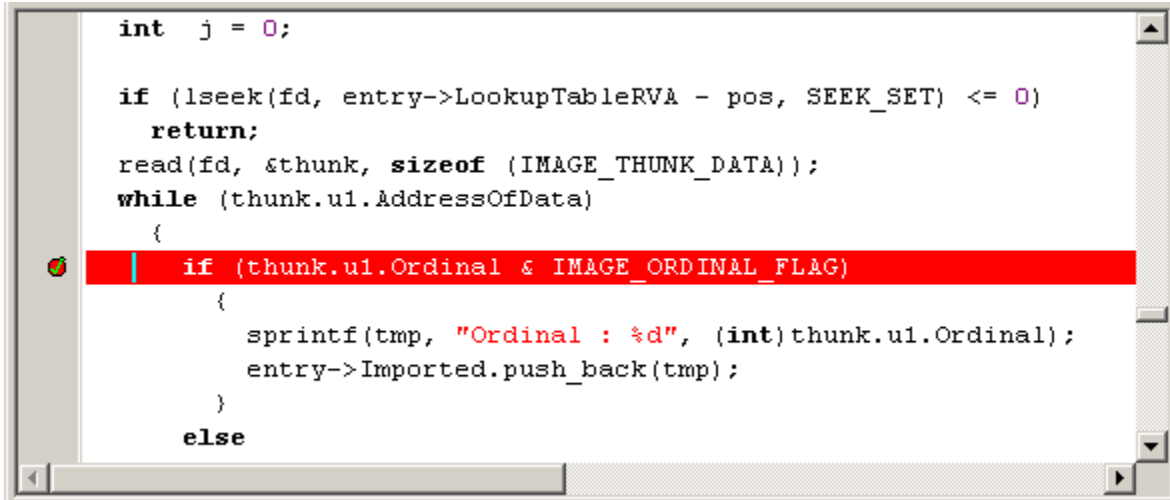
- [Setting Breakpoints in your code](#)ID_BREAKPOINT
- [Stepping in your code](#)ID_STEPDEBUG
- [Displaying variables value and classes/structures members](#)ID_DISPLAYDEBUG
- [Backtracing](#)ID_BACKTRACE
- [Using the CPU Window](#)ID_CPUWINDOW

The output Dev-C++ receives from GDB is displayed in the *Debug Output* sheet at the bottom of Dev-C++. This output is probably only interesting if you are familiar with GDB. You can also send commands directly to it (if you do not know GDB, you can type the *help* command for to have a list) by using the edit box just above the output (see screenshot below).



Setting Breakpoints

You can use breakpoints to pause your program at a certain instruction (line of code). To add a breakpoint, first select the line of code where you want to break by simply positioning the text cursor on it. Now, click on the *Debug* menu, then on *Toggle Breakpoint* (shortcut : Ctrl+F5). Clicking on the gutter (at the left of the editor) in front of your line will have the same effect :

A screenshot of a code editor window. The code is as follows:

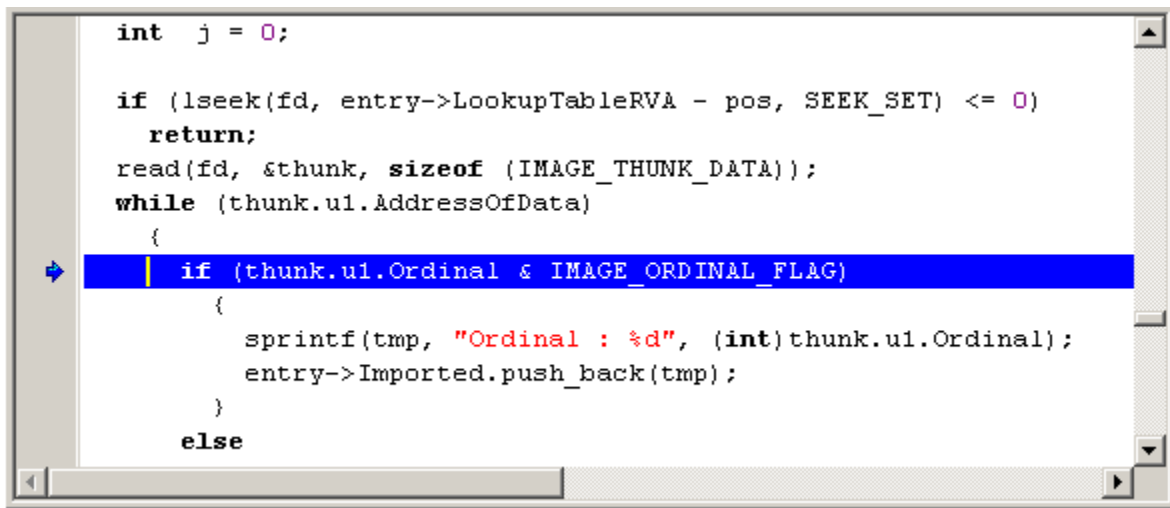
```
int j = 0;

if (lseek(fd, entry->LookupTableRVA - pos, SEEK_SET) <= 0)
    return;
read(fd, &thunk, sizeof (IMAGE_THUNK_DATA));
while (thunk.u1.AddressOfData)
{
    if (thunk.u1.Ordinal & IMAGE_ORDINAL_FLAG)
    {
        sprintf(tmp, "Ordinal : %d", (int)thunk.u1.Ordinal);
        entry->Imported.push_back(tmp);
    }
    else

```

The line `if (thunk.u1.Ordinal & IMAGE_ORDINAL_FLAG)` is highlighted in red. A small red circle with a white dot is in the gutter to the left of this line.


When you will load the debugger `ID_DEBUGPROCESS`, if your program executes the line of code you breakpointed, Dev-C++ will warn you that you're breakpoint was reached by changing the line color to blue :

A screenshot of the same code editor window as above. The code is identical. The line `if (thunk.u1.Ordinal & IMAGE_ORDINAL_FLAG)` is now highlighted in blue. A blue arrow points to the gutter to the left of this line.

You can now examine the data in your program `ID_DISPLAYDEBUG` or step through your program `ID_STEPDEBUG`

Stepping in your program


Once a breakpoint has been reached, you can step into the code of your application in different ways :

- **Next Step**  (shortcut : F7) :

The debugger will step one instruction (line of code) in your program

- **Step Into** (shortcut : Shift+F7) :

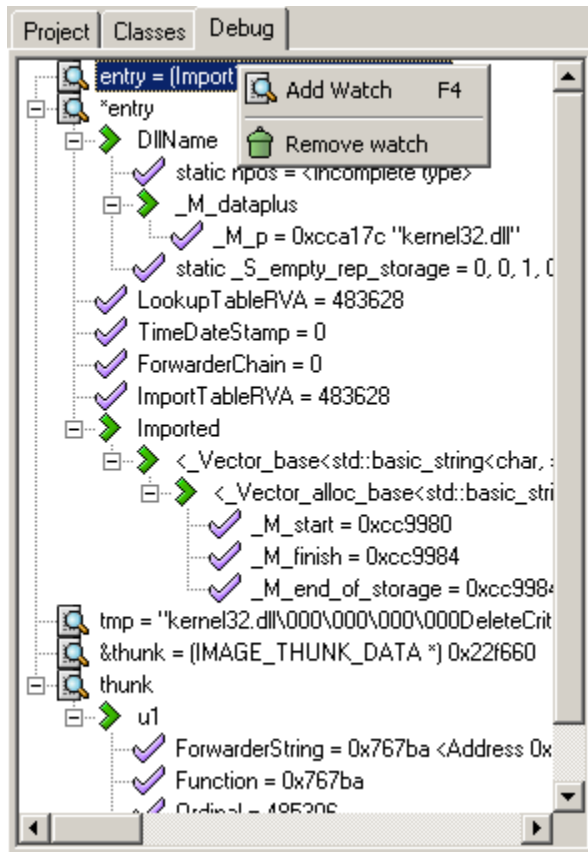
The debugger will step one instruction. If that instruction is a function call it will jump into it.

- **Continue**  (shortcut : Ctrl+F7) :


The debugger will continue the execution of your program until another breakpoint is reached.

Displaying variables value and classes/structures members

One of the other interest of debugging is the possibility to display the value of your variables at a given time. Dev-C++ is able to show you in a convenient way the contents of your classes, strings, structures/unions, arrays and other variables, in the Debug Variable Browser, shown below :



You can display your variables (after you reached a breakpoint) in two different ways :

- Click on the *Add Watch*  (shortcut : F4) button, type the name of your variable in the dialog, and press OK. If you select a word in the current source file and press F4, it will add a watch of the selected text without asking for a variable name.
- Point your mouse over a variable in your source code (if *Watch variable under mouse* is enabled in [Environment Options](#) ID_ENVIRONMENT) and it will be added to the watch list.

Important Notes:

- When using pointers to structures or classes, if you want to display all the members of variable *my_pointer* then you need to watch **my_pointer* ("*" is the value-operator). Watching only *my_pointer* would just display the address contained in *my_pointer*.
- Sometimes the debugger may not know the type of a pointer, and cannot display all of the members of the pointed structure or class. You can bypass this problem by casting your watched variable. For example, if the debugger cannot show the contents of *my_pointer* of type *MyPointer*, you could try adding the watch variable: **(MyPointer *)my_pointer*

Backtracing

Backtracing is the debugging concept that tells you which functions were called before reaching a breakpoint or an interruption (like an access violation).

Let's do a simple test, with the following code:

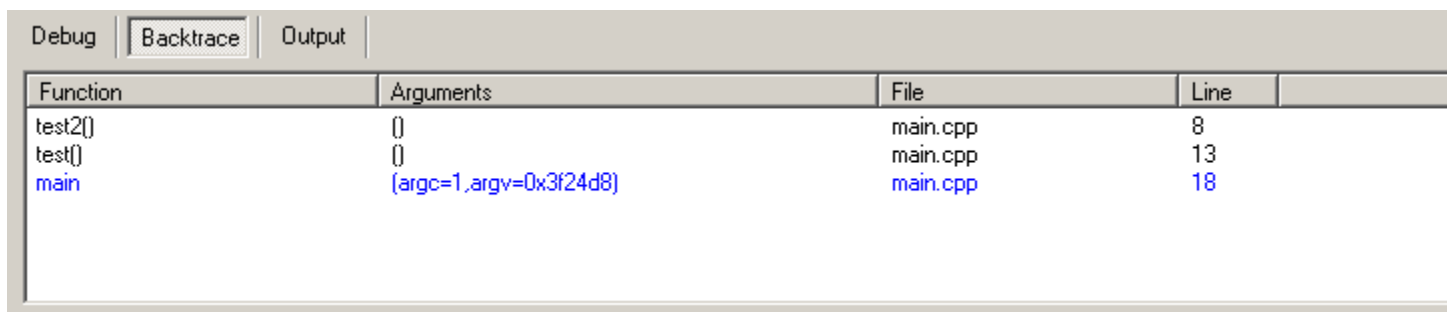
```
#include <stdio.h>

void test2()
{
    printf("hello\n"); /* Put a breakpoint on this line */
}

void test()
{
    test2();
}

int main(int argc, char **argv)
{
    test();
    return 0;
}
```

Put a breakpoint in function `test2()` on the `printf()` statement, and load the debugger. The breakpoint should be reached quite instantly, now go to the *Debug* sheet and click on the *Backtrace* button. The following list should appear:



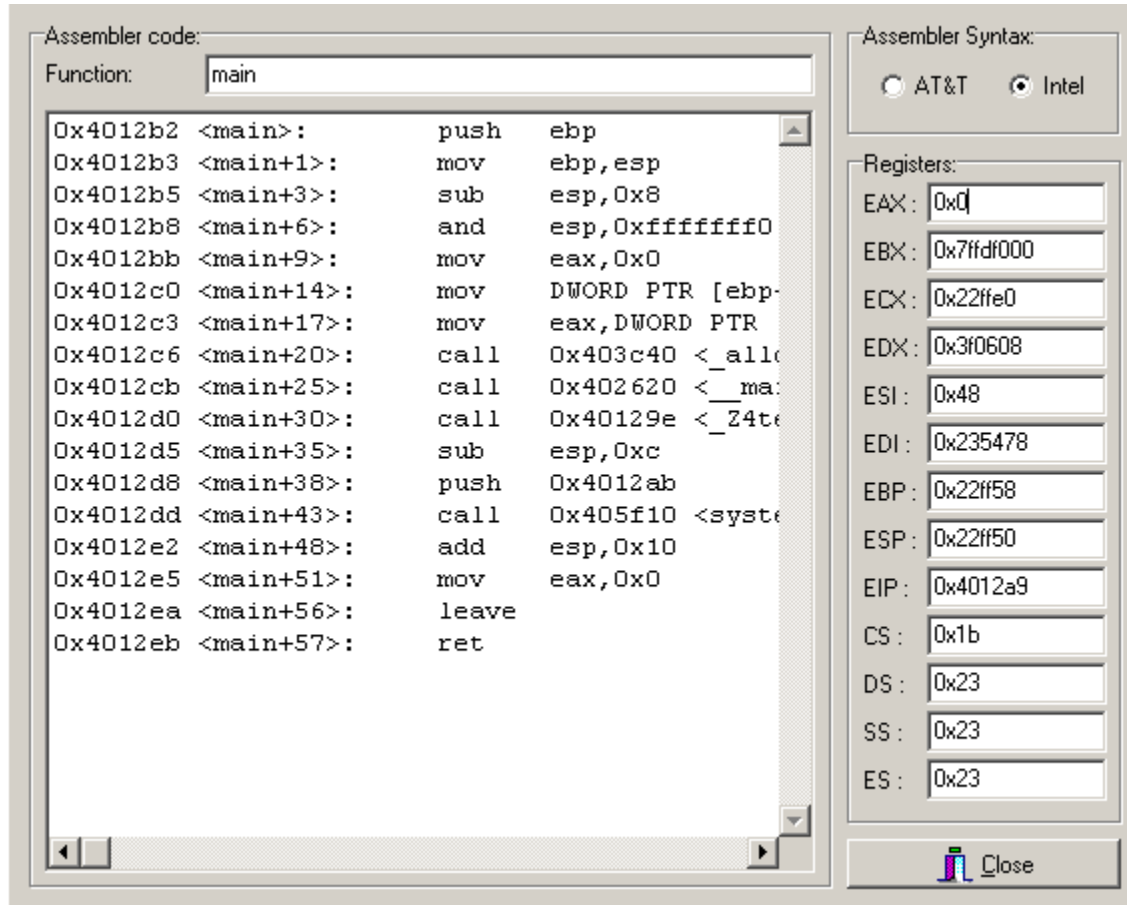
Function	Arguments	File	Line
test2()	()	main.cpp	8
test()	()	main.cpp	13
main	(argc=1,argv=0x3f24d8)	main.cpp	18

This correctly shows the list of functions that have been called (since the start of the program) before reaching the breakpoint. Clicking on a function in this list will bring you to its implementation in your source code.

Using CPU window

Dev-C++ provides a CPU window to expert developers who want access to the status of CPU registers and instructions.

To show the CPU window, wait for a breakpoint or interruption to raise in your program and go to the *Debug* menu, then click *CPU Window*. The following dialog will appear:



On the left, you can find the assembler instructions of the current function. You can display the assembler code of any other function by typing its name in the *Function* field, then pressing the Enter key. You can also select the syntax of the assembler instructions: AT&T or Intel.

On the right, the content of CPU registers is displayed.

Integrated Templates

Dev-C++ uses a template system to be able to create specific projects on the fly. Templates usually contain a set of source files and project file which include the minimal code and all the compiler and linker options, which are necessary to compile the specific type of program.

Getting and installing Packages

to be implemented...

Creating Packages

to be implemented...

Compiler Options

To access the Compiler Options dialog, click on the 'Tools' menu, then on 'Compiler Options'.

Compiler

Compiler set to configure

Here, you can maintain different compiler profiles. Each profile has its own settings and options, so that you can easily switch between different presets (for example, MingW32 and Cygwin). The default compiler that comes bundled with Dev-C++ is MingW32. There are three buttons to add, delete and rename the compiler set.

7

Add the following commands when calling compiler

Here you can specify additional command-line options to be passed to the GCC compiler when compiling your project or file. Check the MingW32 manual (available at the website, www.mingw.org for a list of command-line arguments.

Add these commands to the linker command-line

Here you can specify options to be passed onto the linker (ld.exe). You can specify library files here.

Compile delay

This option is present to provide a delay before compiling. Normally, you will not use this. If you make complaints of the timestamp being invalid, try specifying a delay here.

Use fast but imperfect dependency generation

By default, Dev-C++ will check all files and headers for dependency information, and update the makefile accordingly. If you find that it's taking too much time, you can prevent this by enabling this option.

Settings / C compiler

Support all ANSI standard C programs

Option -ansi: Will attempt to be as ANSI conformant as possible.

Attempt to support some aspects of traditional C pre-processors

Option -traditional-cpp: Will attempt to make the pre-processor behave as traditional ones do.

Inhibit all warning messages

Option -w: No warnings will be displayed.

Settings / C++ compiler

Turn off all access checking

Option -fno-access-control

Accept \$ in identifiers

Option -fdollar-in-identifiers: Will allow \$ to be used in variable and function names. Cannot be used with GCC 3.2.

Use heuristics to compile faster

Option -fsave-memoized

Settings / Code generation

Enable exception handling

Option -fexceptions

Use same size for double and float

Option -fshort-double

Put extra commentary information in the generated assembler

Option -fverbose-asm: The intermediary assembler files generated will have extra comments. Only useful if they're being saved (using the **-S** command-line option).

Settings / Code profiling

Generate profiling info for analysis

Option -pg: Writes extra information into the generated program files to use with the profiler. The profiler lets you see where maximum amount of the program's execution time is being spent, so that when you're optimizing the program, you optimize only the parts that make the difference. This option should be disabled when building retail/final versions. It should be used only in debug builds.

Settings / Linker

Link an Objective C program

Option -lobj

Generate debugging information

Option -g3: Writes debug information into the generate binaries. This lets you debug it with the integrated debugger.

Do not use standard system startup files or libraries

Option -nostdlib

Do not create a console window

Option -mwindows: Instructs GCC to build a Windows GUI application.

Settings / Optimization

Perform a number of minor optimizations

Option -fexpensive-optimizations

Settings / Optimization / Further Optimizations

Here, you can select the optimization level.

Optimize

Option -O1

Optimize more

Option -O2

Best optimization

Option -O3

Directories

Binaries

Specifies the locations of the compiler (executables).

Libraries

Specifies the locations of the library files (*.a).

C includes

Specifies the locations of the headers (*.h) for C programs.

C++ includes

Specifies the locations of the headers (*.h, *.hpp) for C++ programs.

Programs

Here you can specify the filenames of the different compiler executable components.

Environment Options

To access the Environment Options dialog, click on the 'Tools' menu, then on 'Environment Options'.

General

Default to C++ on new project

When you create a new project, it is assumed by default to be in C++. Unselecting this will use C.

Create Backup files

If enabled, whenever you save a source file in Dev-C++, a backup copy will be saved along with it. This backup will be overwritten successively.

Minimize on run

If enabled, Dev-C++ will minimize itself when you execute your program from within it (using the 'Run' command under 'Execute'.

Show toolbars in Full-screen

By default, toolbars are hidden when Dev-C++ is made full-screen. If this option is enabled, they will be shown all the time.

Double-click to open project-manager files

If this option is enabled, you'll need to double-click on the nodes in project-manager to open them in the editor. Otherwise, you would single-click.

Auto-open...

Here, you can choose what files are automatically opened in the editor when you open a project.

Interface

Max files in re-open menu

Here, you can specify the number of files Dev-C++ keeps track of in the Reopen menu. Older files are forgotten first.

Language

Select your language here.

Message Window Tabs

Here, you can select the location of the tabs on the message windows (*Compiler, Resources, Compile Log, Debug, etc.*)

Theme

Select the theme you want Dev-C++ to use.

No splash screen on startup

If enabled, it prevents Dev-C++ for displaying a logo when it is loading.

Use XP theme (WinXP only)

If enabled, it makes Dev-C++ use Windows XP themes (bitmap title-bars, buttons, etc.).

Open/Save dialog style

Here you can select the type of file open/save dialog you want to see.

Files and directories

User's default directory

Specifies the default location that Dev uses (when creating projects, opening, saving, etc.)

Templates

Specifies the location of the Dev-C++ project templates (the ones displayed when you click on 'New Project').

Icon library path

Specifies the location of the icons for use in your projects.

Language Files path

Specifies the location of the Dev-C++ language files.

Splash screen image

You can specify an alternate splash-screen bitmap here.

File associations

Here, you can select one or more file types which Dev-C++ will associate itself with, so that the next time you open an associated

file in Windows Explorer, Dev-C++ will open up automatically.

CVS

CVS program file

Enter the name of the CVS program executable.

Compression level

Specify the compression level to be used.

Use SSH instead of RSH

Use the SSH program to connect

Editor Settings

to be implemented...

Class Browser / Code Completion

to be implemented...

Frequently Asked Questions (FAQ)

Why can't I use *conio.h* functions like `clrsrc()`?

Because `conio.h` is not part of the C standard. It is a Borland extension, and works **only** with Borland compilers (and perhaps some other commercial compilers). Dev-C++ uses [GCC](#), the GNU Compiler Collection, as its compiler. GCC is originally a UNIX compiler, and aims for portability and standards-compliance.

If you really cannot live without them, you can use Borland functions this way:

Include `conio.h` to your source, and add the following file to your project : `C:\Dev-C++\include\conio.c` (where `C:\Dev-C++` is where you installed Dev-C++).

Please note that `conio` support is not complete.

My window keeps closing, how do I change that ?

You can do it this way:

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

```
int main(int argc, char **argv)
{
    printf ("Press ENTER to continue.\n");
    getchar (); /* wait for input */
    return 0;
}
```

Or this way:

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

```
int main(int argc, char **argv)
{
    system ("pause"); /* execute M$-DOS' pause command */
    return 0;
}
```

After linking, I get an error like `C:\DEV-C++\LIB\libmingw32.a(main.o)(.text+0x8e): undefined reference to `WinMain@16'`

You probably haven't declared any `main()` or `WinMain()` function in your program.

How can I provide a `.def` file for my DLL ?

Put in Project Options, Parameters sheet, Linker box : `--def yourfile.def`

I am having strange problems under Windows XP

Try to run Windows Update and make sure that you have the Program Compatibility updates.

How do I enable Debugging mode ?

Go to Compiler Options and click on the Compiler sheet. In the Linker section, put "Yes" to 'Generate debugging information'. Do a 'Rebuild All' and you should be able to debug now.

When I launch Dev-C++ i get the message saying 'WININET.DLL' or 'MSCVRT.DLL' or 'SHFOLDER.DLL' not found

You are missing a Windows DLL, which you can download from Microsoft site or [here](#)

The size of the executable generated is huged, why ?

Be sure you didn't check "Generate debugging information" in Compiler Options, Linker sheet

How to use assembly with Dev-C++ ?

The "GNU as" assembler uses AT&T syntax (not Intel).
Here's an example of such a syntax :

```
// 2 global variables
int AdrIO;
static char ValIO;

void MyFunction(.....)
{
  __asm("mov _AdrIO, %dx"); // loading 16 bits register
  __asm("mov _ValIO, %al"); // loading 8 bits register
/*
Don't forget the underscore _ before each global variable names !
*/
  __asm("mov %ax,%dx"); // AX --> DX
}
```

I am using Windows 98 and I cannot compile

Some users have report that you need to apply several patches to your system. Here is the list of them, they can be found on [Microsoft Windows 98 download site](#)

47569us.exe - labeled as Windows98SE shutdown

dcom98.exe - see also this [page](#)

DX81eng.exe - latest version of DirectX (this is 11MB, and cannot be uninstalled without reinstalling Windows 98. You might want to try this one last in case the other above didn't work, as it should update many parts of the system).

Mailing List / Forum

If you have programming or Dev-C++ specific questions, you can post it on the forum or on the mailing list.

Bloodshed Software / Dev-C++ Forum : <http://bloodshed.net/forum>

Dev-C++ Mailing List : <http://bloodshed.net/devcpp-ml.html>

Dev-C++ FAQ

Common Questions:

1. When I compile my dos program and execute it, Dev-C++ minimizes and then restore in a second but nothing appears?

2. When executing my dos program, it closes automatically. How I can change this ?

3. After linking, i get the error "C:\DEV-C++\LIB\libmingw32.a(main.o)(.text+0x8e): undefined reference to `WinMain@16'

4. When I launch Dev-C++ i get the message saying "WININET.DLL not found"?

5. When I compile a file, I get a message saying "could not find <filename> "

6. The EXE files created are huge. What can i do to reduce the size ?

7. How can i use the OpenGL library and others?

8. When i compile a file that contains references to Windows filename (like <Mydir\myfile.h>), i get an 'unrecognized escape sequence' message?

9. Is there any GUI library or packages available for Dev-C++?

10. I am having problems using Borland specific functions such as clrscr().

11. The toolbars icons are showing incorrectly.

12. It seems i've found a problem/bug that is not specified here. What should i do?

13. How to use assembly (ASM) with Dev-C++?

. When I compile my dos program and execute it, Dev-C++ minimizes and then restore in a second but nothing appears ?

When creating a console application, be sure to uncheck "Do not create a console" in Project Options (when working with source files only uncheck "Create for win32" in Compiler Options).

. When executing my dos program, it closes automatically. How I can change this ?

You can use an input function at the end of you source, like the following example :

```
#include <stdlib.h>
int main()
{
system("PAUSE");
return 0;
}
```

. After linking, i get the error "C:\DEV-C++\LIB\libmingw32.a(main.o)(.text+0x8e): undefined reference to `WinMain@16`"

You probably haven't declared any main() function in your program. Otherwise, try recompiling a second time.

. When I launch Dev-C++ i get the message saying "WININET.DLL not found" ?

If you are missing WININET.DLL on your Windows system, you can download it at:
<http://www.rocketdownload.com/supfiles.htm>

. When I compile a file, I get a message saying "could not find <filename> "

Make sure you included the directory where your include file resides in Project Options.

. The EXE files created are huge. What can i do to reduce the size ?

If you want to reduce your exe file size from 330 Ko to 12 Ko for example, go to compiler options. Then click on the Linker page and uncheck "Generate debug information". This will remove debugging information (if you want to debug, uncheck it). You can also click on Optimization page and check "Best optimization".

. How can i use the OpenGL library and others ?

All the libraries that comes with Mingw reside in the Lib directory. They are all named in the following way: lib*.a

To link a library with your project, just add in Project options, Further option files :

-lopengl32

This is for including the libopengl32.a library. To add any other library, just follow the same syntax:

Type -l (L in lowercase) plus the base name of the library (filename without "lib" and the ".a" extension).

. When i compile a file that contains references to Windows filename (like <\Mydir\myfile.h>), i get a 'unrecognized escape sequence' message ?

The Mingw compiler understands paths in the Unix style (/mydir/myfile.h). Try replacing the \ in the filename by /

. Is there any GUI library or packages available for Dev-C++ ?

You can download extra packages for Dev-C++ at <http://www.bloodshed.net/dev/>

. I am having problems using Borland specific functions such as clrscr()

Include conio.h to your source, and add C:\Dev-C++\Lib\conio.o to "Further Object Files" in Project Options (where C:\Dev-C++ is where you installed Dev-C++)

. The toolbars icons are showing incorrectly.

On some screen resolutions, toolbars icons may show up incorrectly. You should try changing your screen resolution, or disable toolbars from the View menu in Dev-C++

. It seems i've found a problem/bug that is not specified here. What should i do ?

First, you should try doing a "Check for Dev-C++ update" (in Help menu) to know if a new version has come that may correct this problem. If there are no new version or the problem wasn't fixed then please send an email describing the bug to : dev@bloodshed.net

. How to use assembly with Dev-C++ ?

The assembler uses AT&T (not Intel). Here's an example of such a syntax :

```
// 2 global variables
```

```
int AdrIO ;  
static char ValIO ;
```

```
void MyFunction(.....)
```

```
{  
  __asm("mov %dx,_AdrIO") ; // loading 16 bits register  
  __asm("mov %al,_ValIO") ; // loading 8 bits register
```

```
/*  
Don't forget the underscore _ before each global variable names !  
*/
```

```
__asm("mov %dx,%ax") ; // AX --> DX  
}
```

Making and using Templates

Dev-C++ can create and manage template files. Those templates contains information for creating automatically specific projects of your own. Dev-C++ contains 5 default projects, but it lets you design your own too with templates.

Making templates:

To create a new template, click on the File menu and then on "New template file". This will bring you the Template Builder, which creates and registers template files in Dev-C++.

Template Information:

Enter the name, description and category of your template. You can also select an icon, which will be used by the executable generated after compiling a template project.

Editor information: Here you can set the cursor position that will be set when creating a template file. For example, if your templates contains the following default code:

```
int main()
{
```

```
}
```

you should put the following cursor position :

Column = 4 (like for a TAB)

Row = 3 (points to the third line, where the text is empty).

You need to set cursor position for C and C++ codes.

Project information:

This is for setting the default options your template will generate for a project. They are the same as in Project Options.

Code:

Type in the text fields the default C/C++ codes that will be used when creating a project from your template. Default code can be for example:

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

```
int main()
```

```
{
```

```
}
```

When you are ready to save and add your file to the Template list, click on the Save button, and type a filename in the dialog box that follows.

Using templates:

Creating a project from your template is the same as for creating usual projects. Click on the File menu in Dev-C++, then on "New Project...". On the following dialog, click on the "Custom Templates" tab sheet, this will bring you the list of available templates to create a project from. Select a template and press the OK button (or double-click on the selected icon), and a new project will be created using the default options and code as you wrote in your template.

Deleting templates:

In the New Project dialog, select the template you are willing to delete and press the DEL (delete) button on your keyboard.

Compiling and Running

If you want to create the executable of your program and test it, this section will help you on how to do this.

Compiling:

When your sources are ready to be compiled into an executable, click on the Execute menu, then on Compile. The compilation will start and if it is successful (no errors in your program), then an executable with the name of your project will be created in your project's directory.
If you got errors after compile, the errors will be shown in the Compiler sheet.

Running:

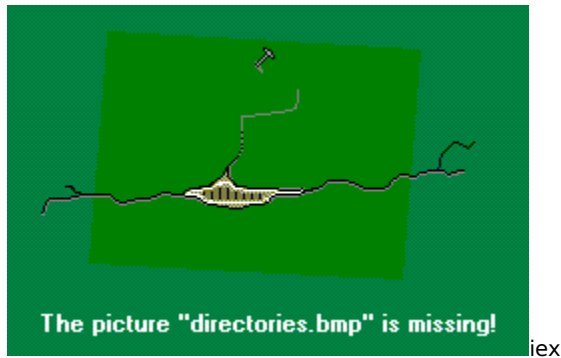
If your project has been successfully compiled, you can execute it by clicking on the Execute menu, then on Run.

Compiler Options

This section is about to explain you all the options available in Dev-C++ when running the compiler. The Compiler options dialog is available by clicking on the Options menu, then on "Compiler options".

Most of the parameters description have been taken from the GCC help file.

Directories page:

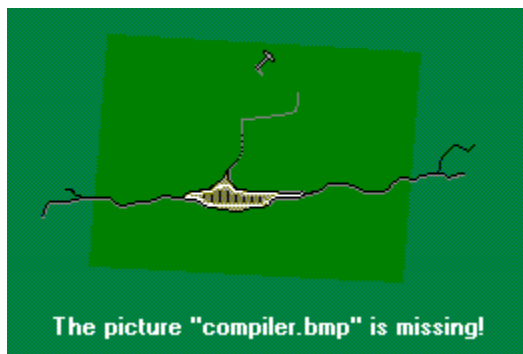


You can add multiple directories to be searched for include files during compilation by checking the appropriate check box, and by typing your directory in the edit field (separate pathnames with a semicolon ";").

Other compiler-specific parameters can be added by checking the check box "Add the following commands...", and by typing your commands in the edit field (separate commands by spaces). These commands can be found at Mingw32 web site.

You can also change/add the different directories needed by Dev-C++ (separate folders with a semicolon, but only one Bin directory can be specified).

C/C++ compiler page:



- Support all ANSI C standard programs (*compiler parameter: -ansi*):

This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, and it disables recognition of C++ style `//` comments.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite `-ansi`. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with `-ansi`. Alternate predefined macros such as `__unix__` and `__vax__` are also available, with or without `-ansi`.

The `-ansi` option does not cause non-ANSI programs to be rejected gratuitously. For that, `-pedantic` is required in addition to `-ansi`. See Warning Options.

The macro `__STRICT_ANSI__` is predefined when the `-ansi` option is used. Some header files may notice this macro

and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

The functions `alloca`, `abort`, `exit`, and `_exit` are not builtin functions when `-ansi` is used.

- Try to support some aspects of the traditional C preprocessor (*compiler parameter: -traditional*):
All extern declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.

The newer keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)

Comparisons between pointers and integers are always allowed. Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.

Out-of-range floating point literals are not an error.

Certain constructs which ANSI regards as a single invalid preprocessing number, such as `0xe-0xd`, are treated as expressions instead.

String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of `-fwritable-strings`.)

Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.

The character escape sequences `\x` and `\a` evaluate as the literal characters `x` and `a` respectively. Without `-traditional`, `\x` is a prefix for the hexadecimal representation of a character, and `\a` produces a bell.

In C++ programs, assignment to this is permitted with `-traditional`. (The option `-fthis-is-variable` also has this effect.)

You may wish to use `-fno-builtin` as well as `-traditional` if your program uses names that are normally GNU C builtin functions for other purposes of its own.

You cannot use `-traditional` if you include any header files that rely on ANSI C features. Some vendors are starting to ship systems with ANSI C header files and you cannot use `-traditional` on such systems to compile files that include any system headers.

- **Show all warnings** (compiler parameter : `-w`):
This will tell the compiler to show warnings all the warnings.

- **Turn off all access checking** (compiler parameter : `-fno-accesschecking`):
Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.

- **Accept \$ in identifiers** (compiler parameter : `-fdollars-in-identifier`):
Accept `$` in identifiers. You can also explicitly prohibit use of `$` with the option `-fno-dollars-in-identifiers`. (GNU C allows `$` by default on most target systems, but there are a few exceptions.) Traditional C allowed the character `$` to form part of identifiers. However, ANSI C and C++ forbid `$` in identifiers.

- **Use heuristics to compile faster** (compiler parameter : `-fsave-memorized`):
Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.

The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this:

```
cout << "This " << p << " has " << n << " legs.\n";
```

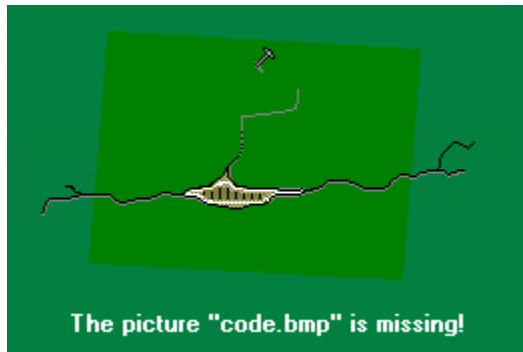
makes six passes through all three steps. By using a software cache, a "hit" significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms which must be implemented, and so incurs its own overhead. `-fmemoize-lookups` enables the software cache.

Because access privileges (visibility) to members and member functions may differ from one function context to the next, G++ may need to flush the cache. With the `-fmemoize-lookups` flag, the cache is flushed after every function

that is compiled. The `-fsave-memoized` flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

The code that implements these flags has rotted; you should probably avoid using them.

Code generation/Optimization page:



- Enable exception handling (compiler parameter: `-fexceptions`):

Enable exception handling, and generate extra code needed to propagate exceptions. If you do not specify this option, GNU CC enables it by default for languages like C++ that normally require exception handling, and disabled for languages like C that do not normally require it. However, when compiling C code that needs to interoperate properly with exception handlers written in C++, you may need to enable this option. You may also wish to disable this option if you are compiling older C++ programs that don't use exception handling.

- Use the same size as double as for float (*compiler parameter: `-fshortdouble`*)

- Put extra information in generated assembly code (*compiler parameter: `-fverbose-asm`*):

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

`-fno-verbose-asm`, the default, causes the extra information to be omitted and is useful when comparing two assembler files.

- Optimize (compiler parameter: `-O`):

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without `-O`, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without `-O`, the compiler only allocates variables declared register in registers. The resulting compiled code is a little worse than produced by PCC without `-O`.

With `-O`, the compiler tries to reduce code size and execution time.

When you specify `-O`, the compiler turns on `-fthread-jumps` and `-fdefer-pop` on all machines. The compiler turns on `-fdelayed-branch` on machines that have delay slots, and `-fomit-frame-pointer` on machines that can support debugging even without a frame pointer. On some machines the compiler also turns on other flags.

- Best optimization (compiler parameter: `-O3`):

Optimize even more. GNU CC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code.

`-O3` turns on all optional optimizations like for loop unrolling and function inlining. It also turns on the `-fforce-mem` option on all machines and frame pointer elimination on machines where doing so does not interfere with debugging and also turns on the `inline-functions` option.

- **Perform a number of minor optimizations** (compiler parameter: `-fexpensive-optimizations`):
Perform a number of minor optimizations that are relatively expensive.

Linker Page:



- Link an objective C program (*compiler parameter: `-lobjc`*):
You need this special case of the `-l` option in order to link an Objective C program.

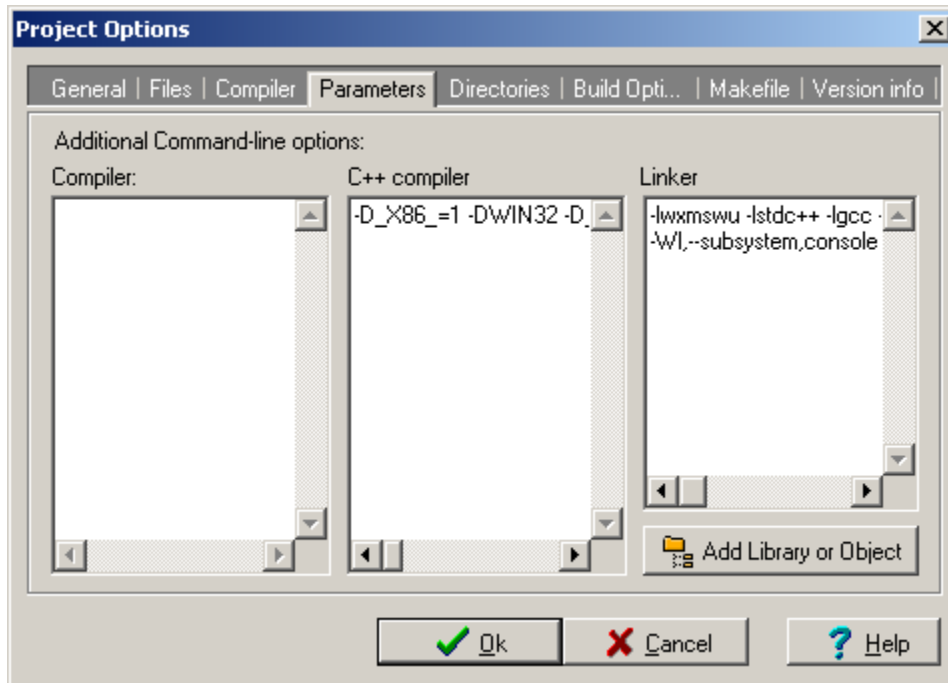
- Generate debugging information (compiler parameter: `-ggdb`) :
Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF 2, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

- Do not use standard system files or libraries
Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker.

One of the standard libraries bypassed by `-nostdlib` and `-nodefaultlibs` is `libgcc.a`, a library of internal subroutines that GNU CC uses to overcome shortcomings of particular machines, or special needs for some languages. In most cases, you need `libgcc.a` even when you want to avoid other standard libraries. In other words, when you specify `-nostdlib` or `-nodefaultlibs` you should usually specify `-lgcc` as well. This ensures that you have no unresolved references to internal GNU CC library subroutines. (For example, `__main`, used to ensure C++ constructors will be called; see `collect2`.)

- Compile for Win32 – no console (*compiler parameter : `-mwindows`*):
This is used when working on source files (for projects, use "Do not create a console" in Project Options). It will tell the linker to create a windows program with no dos console.

Help v1.02B



Bloodshed Software:

Bloodshed Software is a non-profit organization devoted to serve programmers and internet users by providing them free softwares and free internet services.

About Dev-C++:

Dev-C++ is a full-featured integrated development environment (IDE), which is able to create Windows (GUI, console, DLLs...) C/C++ programs using the Mingw compiler system or theoretically any GCC based compiler.

Bloodshed Software: <http://www.bloodshed.net/>

About MinGW:

MinGW is a collection of header files and import libraries that allow one to use GCC and produce native Windows32 programs that do not rely on any 3rd-party DLLs. The current set of tools include GNU Compiler Collection (GCC), GNU Binary Utilities (Binutils), GNU debugger (Gdb), GNU make, and a assorted other utilities. We are currently working on creating a complete set of Mingw-hosted GNU toolchain, and looking for volunteers.

At the basic level, MinGW is a set of include files and import libraries that allow a console-mode program to use Microsoft's standard C runtime library MSVCRT.DLL (available on all NT systems, and on all stock Win9x after the original Windows 95 release (for which it can be installed separately). Using this basic runtime, you can write console-mode ANSI compliant programs using GCC and use some of the extensions offered by MS C runtime, but cannot use the features provided by the Windows32 API. The next critical piece is the w32api package, which is a set of includes and import libraries to enable the use of Windows32 API, and combined with the basic runtime, you (potentially) have full access to both the C Runtime (CRT) and Windows32 API functionality. Please see the licensing information on the various pieces.

See [here](#) for information on how Mingw differs from Red Hat/Cygnus Cygwin and AT&T UWIN.

If you see any errors on this help file, please contact either [Kip](#), or [Colin](#).

Credits:

[Colin Laplace](#) Hongli Lai Mumit Khan Jan Jaap van Main Development... IDE updates, icons, templates... Mingw

der Heidjen Colin Hendrix GNU coders Kipling Warner

compiler Mingw compiler Mingw compiler Mingw compiler Splash screen, Help file, critic...

Dev-C++ IDE development:

Colin Laplace and Mike Berg : Main development

Kipling Warner: Help System, Splash Screen...

Mingw compiler system development:

Mumit Khan, J.J. Van Der Heidjen, Colin Hendrix and GNU coders...

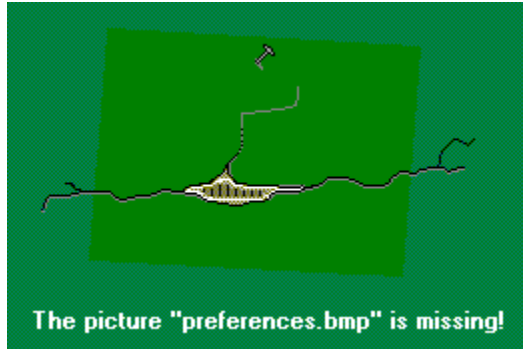
VCL component authors (used in Dev-C++ IDE):

SynEdit : Powerful Editor, with syntax highlighting, exporting... <http://synedit.sourceforge.net>

Environment Options

The environment options are described for the following sheets :

Preferences:



Default directory:You can select a directory that will be automatically use when creating, opening and saving files.

Create backup file:Check this if you want Dev-C++ to create a backup of the sources when saving.

Auto-save desktop and editor's position:If you want to use the same size of the editor and main window every time.

Save toolbars availability:Dev-C++ will restore the toolbars depending on latest session.

Auto-Arrange windows: Tile windows often.

Do not show project manager:Check this if you don't want to work with the Project Manager.

Make compile result window stay on top: After compiling, if this option is checked the compile window result will remain stay on top.

Minimize during execution:Dev-C++ will minimize when executing your program.

Show exit code after running:Tells Dev-C++ to show the exit code of your program after executing it.

Give the following parameters when executing a compiled project:Use this to call your program with parameters, in case it needs some.

Editor:



Editor font, size and background color:You can set these options to work with what you are used to.

Show lines number:This shows line numbers in the left the editor.

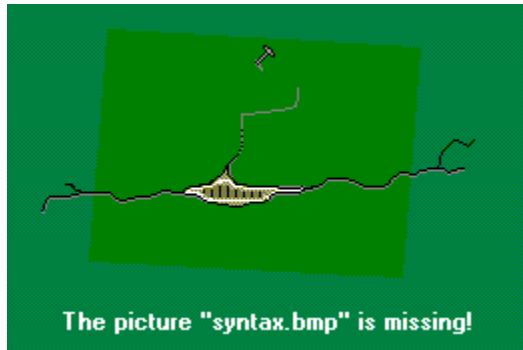
Auto-Indent: Use Dev-C++ tab settings (intelligent tabs).

Tab indent: Select Tab size. You can only use this feature if "Auto-Indent" is unchecked.

Show hint when scrolling: This will show a hint with the current line number when scrolling.

Do not show scrollbars: Check it if you don't want scrollbars in the editor.

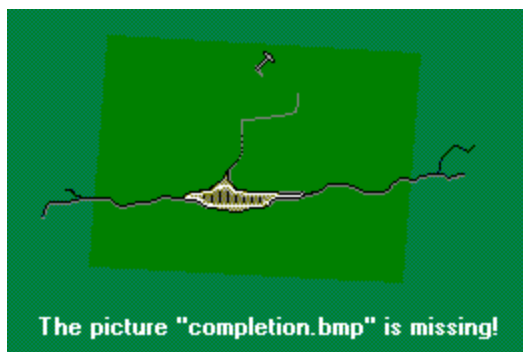
Syntax colors:



You can modify the different syntax color of the C and C++ grammar, by first clicking on the type you would like to change. Then, select a color by clicking, as well as a text attributes (optional).

You will be available to view your change in the editor in this sheet.

Code completion:



Code completion is for speeding up your coding time. By pressing Ctrl+Space in the editor, a dialog will open containing a list of your usual coding text. Then you will be able to select a code and after pressing Enter it will be automatically inserted in the editor. You can edit them by modifying the text field on the sheet.

Misc.:



Assign .dev files with Dev-C++:This is set by default. It allows you to open Dev-C++ project files by double-clicking in the Windows Explorer.

Assign .c and .cpp files with Dev-C++:This is set by default. It allows you to open .c and .cpp files by double-clicking in the Windows Explorer.

Buttons:By clicking on these buttons you will be able to reset the default options of Dev-C++ and remove files associations.

Default code when creating new source files:Type in the editor the code you would like to have when creating new source files.

Help Files

A tutorial for making projects as well as the complete Standard Template Library guide are available by clicking on the Help menu in Dev-C++.

These files are a good way to learn more about Dev-C++ and the C++ programming language. Help files for the Mingw compiler are available at Mingw or Mumin Khan's site:

<http://www.mingw.org/>

<http://www.xraylith.wisc.edu/~khan/software/gnu-win32/>

Mailing list

The purpose of this mailing list is to ask and answer questions about Dev-C++ and C++ programming. Please note that you may receive a few messages from the list everyday. You can subscribe to the Dev-C++ mailing list by going to the following address, where you will get all the information needed:

<http://www.bloodshed.net/devcpp-ml.html>

Contacting Authors

Suggestions, remarks are always welcome, but please try to look at the FAQ or the Bloodshed forum (<http://www.bloodshed.net/forum>) before asking help on a problem (90 % of problems are described there, and you can also find the last updated FAQ at (<http://www.bloodshed.net/dev/faq.html>)).

Colin Laplace at haiku@bloodshed.net

Hongli Lai at h.lai@chello.nl

Yiannis Mandravellos at mandrav@supergoal.gr

Bloodshed Software homepage : <http://www.bloodshed.net>

Mingw homepage : <http://www.mingw.org/>

Step 1

This tutorial covers step-by-step the creation of a C++ Windows project, as well as managing its options.

To create a new project, click on the File menu, then on "New project". In the project type dialog, select "Windows application" and make sure the C++ button is checked. Click on OK to continue.

[Step 2](#)Topic38

Step 2

Type a name for your project in the next dialog, like "Project 1" and click on OK. You will have then to save your project in a directory. You can save it for example in c:\My Documents\ and then click on Save. All the necessary files will be created in the directory you selected and your project will be opened in Dev-C++, with a new source file inside (the code inside it is the base for a win32 program).

Step 3Topic39

Step 3

Now we are going to compile the project, and create a Windows executable.

Click on the Execute menu, and click on "Compile". A dialog will ask you where do you want to save your file. Select a directory as well as a filename for your file and click on Save. A windows titled "Please wait while compiling..." will appear. Dev-C++ is currently compiling the file. If you don't get any compile errors, you can click on the Execute button to launch the executable of your program.

If any error occurs, it will be displayed in the Compiler output panel (at the bottom of Dev-C++). You can then double-click on it to go to the error in your source.

Step 4Topic40

Step 4

You can now do the following things with your project:

- Create a new file : click on Project menu, then "New unit in project"
- Add a file to your project : click on Project menu, then "Add unit to project"
- Remove a file from your project : right-click on the file you would like to remove in the Project Manager, and click on "Remove from project" (this can also be accessed by the Project menu)
- Rename a file in your project : right-click on the file you would like to rename in the Project Manager, and click "Rename file"
- Set Project Options : follow [Step 5](#)
- Modify the resource file of your program : follow [Step 6](#)

Step 5

Managing project options:

To open the Project Options dialog, click on the Project menu, then on "Project options".

Options are described here:

-Project icon : you can modify the default icon for your application, either by loading an existing icon from your hard disk (click on "Load Icon...", or either by selecting an icon in the Icon Library (click on "Icon Library").

-Object files : Add object files to be linked with your project, as well as libraries and compiler options.

-Extra compiler options : You can add here compiler parameters that will be used when compiling.

-Extra include directories : You can specify here your project's include directories.

-Resource files : Add further resource file to your project

-Do not create a console : Check this if you are willing to make a Windows program

-Compile for C++ : Compile C++ files or not

-Create a DLL : Check this is you want to create a DLL instead of an EXE file.

Step 6Topic42

Step 6

Modifying the resource file:

To open the Resource Editor, click on the Project menu, then on "Edit resource file". Then, you will be able to modify your resource file. The Resource Editor can also easily add menus, bitmaps, fonts and icons to your resource file, by clicking on the appropriate buttons in the dialog. When creating menus, you should take a look at the WinMenu example included with Dev-C++ to know how to set the menu in your program.

#\$Using integrated debugger

todo

Introduction to C Programming

So you want to learn C? We hope to provide you with an easy step by step guide to programming in C. The course is split up into several sections, or lessons, which include C example programs for you to demonstrate what has been taught. Although the ordering of the sections does not have to be strictly followed, the sections become progressively more involved and assume background knowledge attained from previous sections. Good Luck!

Before you start:

1. Please read this Introduction.
2. It is a long course and will take you quite a while to complete. If you use the Hotlist or Bookmark feature of your browser you will be able to return to the place where you left off at or to return to a particular section.
3. This tutorial should be viewable on any WWW browser - if you have any problems please let us know!.

The Course Section Topics:

1. Overview of C.
 - a. Why use C?
 - b. Uses of C
 - c. A Brief History of C
 - d. C for Personal Computers
2. Running C Programs.
 - a. Using Microsoft C.
 - b. Unix System.
3. Structure of C Programs.
 - a. C's Character Set
 - b. The form of a C Program
 - c. The layout of C Programs
 - d. Preprocessor Directives
4. Your First Program.
 - a. Commenting Programs.
5. Data Types - Part I.
 - a. Integer Number Variables.
 - b. Decimal Number Variables.
 - c. Character Variables.
 - d. Assignment Statement.
 - e. Arithmetic Ordering.
 - f. Something To Declare.
6. Input and Output
 - a. printf.
 - b. The % Format Specifiers.
 - c. Formatting Your Output.
 - d. scanf.
7. Control Loops
 - a. The while and do while Loops.
 - b. Conditions, or Logical Expression.
 - c. The for Loop.
8. Conditional Execution
 - a. Program Control - if , if else etc..
 - b. Using break and continue Within Loops.
 - c. Select Paths with switch.
9. Structure and Nesting
10. Functions and Prototypes

- a.Functions - C's Building Blocks.
- b.Functions and Local Variables.
- c.Getting the Value of Variables into Functions.
- d.Functions and Prototypes.
- e.What is ANSI C?.
- f.Standard Library Functions.

11.Data Types - Part II

- a.Global Variables.
- b.Constant Data Types.

12.Arrays

13.Pointers

- a.Point To Point.
- b.Swap Shop.
- c.Pointers Linked To Arrays.

14.Strings

- a.Stringing Along.
- b.As easy as... B or C?.
- c.A Sort OF Bubble Program.

15.Structures

- a.Defining A New Type.
- b.Structures and Functions.
- c.Pointers To Structures.
- d.Malloc.
- e.Structures and Linked Lists.
- f.Structures and C++.
- g.Header Files.

16.File Handling

- a.Stream Files.
- b.Text File Functions.
- c.Binary File Functions.
- d.File System Functions.
- e.Command Line Parameters.

17.Recommended Books

18.Appendix: C's functions

You've now reached the end of this online tutorial. We have covered a lot of ground - but this has been a first course in C and there is still plenty to learn. However, as long as you keep in mind that C is an essentially simple language and how new features are built from this simplicity you shouldn't have many problems.

You also need to be aware of the fact that C is a very low-level language and as a result allows programmers to confuse data types and muck around with the bit patterns of the data in a way that higher level languages would disown! You probably need to make sure that you understand binary and the way that values are represented to get the best from C.

Overview of C

Objectives:

This section is designed to give you a general overview of the C programming language. Although much of this section will be expanded in later sections it gives you a taste of what is to come.

Why use C?:

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient compilers are available for machines ranging in power from the Apple Macintosh to the Cray supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

- the portability of the compiler;
- the standard library concept;
- a powerful and varied repertoire of operators;
- an elegant syntax;
- ready access to the hardware when needed;
- and the ease with which applications can be optimized by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. Fortran) provides everything the programmer might want to do already built into the language. A low level language (e.g. assembler) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Uses of C

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

In recent years C has been used as a general-purpose language because of its popularity with programmers. It is not the world's easiest language to learn and you will certainly benefit if you are not learning C as your first programming language! C is trendy (I nearly said sexy) - many well established programmers are switching to C for all sorts of reasons, but mainly because of the portability that writing standard C programs can offer.

A Brief History of C:

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. BCPL and B are "typeless" languages whereas C

provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

A Rough Guide to Programming Languages is available on-line for those of you that are interested.

C for Personal Computers:

With regards to personal computers Microsoft C for IBM (or clones) PC's. and Borlands C are seen to be the two most commonly used systems. However, the latest version of Microsoft C is now considered to be the most powerful and efficient C compiler for personal computers.

Running C Programs

Objectives:

Having read this section you should be able to:

1. Edit, link and run your C programs

This section is primarily aimed at the beginner who has no or little experience of using compiled languages. We cover the various stages of program development. The basic principles of this section will apply to whatever C compiler you choose to use, the stages are nearly always the same

The Edit-Compile-Link-Execute Process:

Developing a program in a compiled language such as C requires at least four steps:

1. editing (or writing) the program
2. compiling it
3. linking it
4. executing it

We will now cover each step separately.

Editing:

You write a computer program with words and symbols that are understandable to human beings. This is the edit part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the source file (you can read it with the TYPE command in DOS or the cat command in unix). The custom is that the text of a C program is stored in a file with the extension .c for C programming language

Compiling:

You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit (for example, the 80*87 microprocessor). This process produces an intermediate object file - with the extension .obj, the .obj stands for Object.

Linking:

The first question that comes to most people's minds is Why is linking necessary? The main reason is that many compiled languages come with library routines which can be added to your program. These routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (stdio.h) so even the most basic program will require a library function. After linking the file extension is .exe which are executable files.

Executable files:

Thus the text editor produces .c source files, which go to the compiler, which produces .obj object files, which go to the linker, which produces .exe executable file. You can then run .exe files as you can other applications, simply by typing their names at the DOS prompt or run using windows menu.

Using Microsoft C:

Edit stage:

Type program in using one of the Microsoft Windows editing packages.

Compile and link:

Select Building from Make menu. Building option allows you to both compile and link in the same option.

Execute:

Use the Run menu and select Go option.

Errors:

First error highlighted. Use Next Error from Search menu for further errors if applicable.

If you get an error message, or you find that the program doesn't work when you finally run it (at least not in the way you anticipated) you will have to go back to the source file - the .c file - to make changes and go through the whole development process again!

Unix systems:

The University's central irix Service is a Silicon Graphics Inc. Challenge XL system which runs a Unix-like operating system called IRIX. The basic information to run a C program on this system is covered in document HT.SI.05 - How To... Run C Programs On The irix Service. Although this document refers to the IRIX operating system many of the command options will be common to all Unix systems.

On all Unix systems further help on the C compiler can be obtained from the on-line manual. Type

`man cc`

on your local Unix system for more information.

Please note that Unix is a case sensitive operating system and files named `firstprog.c` and `FIRSTPROG.c` are treated as two separate files on these system. By default the Unix system compiles and links a program in one step, as follows:

```
cc firstprog.c
```

This command creates an executable file called `a.out` that overwrites any existing file called `a.out`. Executable files on Unix are run by typing their name. In this case the program is run as follows:

```
a.out
```

To change the name of the executable file type:

```
cc -o firstprog.c
```

This produces an executable file called `firstprog` which is run as follows:

```
firstprog
```


Structure of C Programs

Objectives:

Having completed this section you should know about:

- 1.C's character set
- 2.C's keywords
- 3.the general structure of a C program
- 4.that all C statement must end in a ;
- 5.that C is a free format language
- 6.all C programs use header files that contain standard library functions.

C's Character Set:

C does not use, nor requires the use of, every character found on a modern computer keyboard. The only characters required by the C Programming Language are as follows:

```
A - Z
a - z
0 - 9
space . , ; ' $ "
# % & ! _ { } [ ] ( ) < > |
+ - / * =
```

The use of most of this set of characters will be discussed throughout the course.

The form of a C Program:

All C programs will consist of at least one function, but it is usual (when your experience grows) to write a C program that comprises several functions. The only function that has to be present is the function called main. For more advanced programs the main function will act as a controlling function calling other functions in their turn to do the dirty work! The main function is the first function that is called when your program executes.

C makes use of only 32 keywords which combine with the formal syntax to form the C programming language. Note that all keywords are written in lower case - C, like UNIX, uses upper and lowercase text to mean different things. If you are not sure what to use then always use lowercase text in writing your C programs. A keyword may not be used for any other purposes. For example, you cannot have a variable called auto.

The layout of C Programs:

The general form of a C program is as follows (don't worry about what everything means at the moment - things will be explained later):

```
preprocessor directives
global declarations
main()
{
    local variables to function main ;
    statements associated with function main ;
}
f1()
{
    local variables to function 1 ;
    statements associated with function 1 ;
}
f2()
{
    local variables to function f2 ;
    statements associated with function 2 ;
}
.
```

.
.
etc

Note the use of the bracket set () and {}. () are used in conjunction with function names whereas {} are used as to delimit the C statements that are associated with that function. Also note the semicolon - yes it is there, but you might have missed it! a semicolon (;) is used to terminate C statements. C is a free format language and long statements can be continued, without truncation, onto the next line. The semicolon informs the C compiler that the end of the statement has been reached. Free format also means that you can add as many spaces as you like to improve the look of your programs.

A very common mistake made by everyone, who is new to the C programming language, is to miss off the semicolon. The C compiler will concatenate the various lines of the program together and then tries to understand them - which it will not be able to do. The error message produced by the compiler will relate to a line of you program which could be some distance from the initial mistake.

Preprocessor Directives:

C is a small language but provides the programmer with all the tools to be able to write powerful programs. Some people don't like C because it is too primitive! Look again at the set of keywords that comprises the C language and see if you can find a command that allows you to print to the computer's screen the result of, say, a simple calculation. Don't look too hard because it doesn't exist.

It would be very tedious, for all of us, if everytime we wanted to communicate with the computer we all had to write our own output functions. Fortunately, we do not have to. C uses libraries of standard functions which are included when we build our programs. For the novice C programmer one of the many questions always asked is does a function already exist for what I want to do? Only experience will help here but we do include a function listing as part of this course.

All programs you will write will need to communicate to the outside world - I don't think I can think of a program that doesn't need to tell someone an answer. So all our C programs will need at least one of C's standard libraries which deals with standard inputting and outputting of data. This library is called `stdio.h` and it is declared in our programs before the main function. The `.h` extension indicates that this is a header file.

I have already mentioned that C is a free format language and that you can layout your programs how you want to using as much white space as you like. The only exception are statements associated with the preprocessor.

All preprocessor directives begin with a # and the must start in the first column. The commonest directive to all C programs is:

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

Note the use of the angle brackets (< and >) around the header's name. These indicate that the header file is to be looked for on the system disk which stores the rest of the C program application. Some text books will show the above statement as follows:

```
#include "stdio.h"
```

The double quotes indicate that the current working directory should be searched for the required header file. This will be true when you write your own header files but the standard header files should always have the angle brackets around them.

NOTE: just to keep you on your toes - preprocessor statements, such as include, DO NOT use semi-colons as delimiters! But don't forget the # must be in the first column.

Thats enough background to C programs - lets get on with our first program which will start to bring together some of the ideas outlined above.

Your First Program

Objectives:

Having read this section you should have an understanding of:

- 1.a preprocessor directive that must be present in all your C programs.
- 2.a simple C function used to write information to your screen.
- 3.how to add comments to your programs

Now that you've seen the compiler in action it's time for you to write your very own first C program. You can probably guess what it's going to be - the program that everyone writes just to check they understand the very, very, very basics of what is going on in a new language.

Yes - it's the ubiquitous "Hello World" program. All your first program is going to do is print the message "Hello World" on the screen.

The program is a short one, to say the least. Here it is:

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return 0;
}
```

The first line is the standard start for all C programs - main(). After this comes the program's only instruction enclosed in curly brackets { }. The curly brackets mark the start and end of the list of instructions that make up the program - in this case just one instruction.

Notice the semicolon marking the end of the instruction. You might as well get into the habit of ending every C instruction with a semicolon - it will save you a lot of trouble! Also notice that the semicolon marks the end of an instruction - it isn't a separator as is the custom in other languages.

If you're puzzled about why the curly brackets are on separate lines I'd better tell you that it's just a layout convention to help you spot matching brackets. C is very unfussy about the way you lay it out. For example, you could enter the Hello World program as:

```
main(){printf("Hello World\n");}
```

but this is unusual.

The printf function does what its name suggest it does: it prints, on the screen, whatever you tell it to. The "\n" is a special symbols that forces a new line on the screen.

OK, that's enough explanation of our first program! Type it in and save it as Hello.c. Then use the compiler to compile it, then the linker to link it and finally run it. The output is as follows:

```
Hello World
```

Add Comments to a Program:

A comment is a note to yourself (or others) that you put into your source code. All comments are ignored by the compiler. They exist solely for your benefit. Comments are used primarily to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it. You can also use a comment to temporarily remove a line of code. Simply surround the line(s) with the comment symbols.

In C, the start of a comment is signaled by the /* character pair. A comment is ended by */. For example, this is a syntactically correct C comment:

```
/* This is a comment. */
```

Comments can extend over several lines and can go anywhere except in the middle of any C keyword, function

name or variable name. In C you can't have one comment within another comment. That is comments may not be nested. Lets now look at our first program one last time but this time with comments:

```
main() /* main function heading */
{
    printf("\n Hello, World! \n"); /* Display message on */
}                                  /* the screen */
```

This program is not large enough to warrant comment statements but the principle is still the same.

Data Types

Objectives:

Having read this section you should be able to:

1. declare (name) a local variable as being one of C's five data types
2. initialise local variables
3. perform simple arithmetic using local variables

Now we have to start looking into the details of the C language. How easy you find the rest of this section will depend on whether you have ever programmed before - no matter what the language was. There are a great many ideas common to programming in any language and C is no exception to this rule.

So if you haven't programmed before, you need to take the rest of this section slowly and keep going over it until it makes sense. If, on the other hand, you have programmed before you'll be wondering what all the fuss is about. It's a lot like being able to ride a bike!

The first thing you need to know is that you can create variables to store values in. A variable is just a named area of storage that can hold a single value (numeric or character). C is very fussy about how you create variables and what you store in them. It demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it.

In this section we are only going to be discussing local variables. These are variables that are used within the current program unit (or function) in a later section we will look at global variables - variables that are available to all the program's functions.

There are five basic data types associated with variables:

- int - integer: a whole number.
- float - floating point value: ie a number with a fractional part.
- double - a double-precision floating point value.
- char - a single character.
- void - valueless special purpose type which we will examine closely in later sections.

One of the confusing things about the C language is that the range of values and the amount of storage that each of these types takes is not defined. This is because in each case the 'natural' choice is made for each type of machine. You can call variables what you like, although it helps if you give them sensible names that give you a hint of what they're being used for - names like sum, total, average and so on. If you are translating a formula then use variable names that reflect the elements used in the formula. For example, $2\pi r$ (that should read as "2 pi r" but that depends upon how your browser has been set-up) would give local variables names of pi and r. Remember, C programmers tend to prefer short names!

Note: all C's variables must begin with a letter or a "_" (underscore) character.

Integer Number Variables:

The first type of variable we need to know about is of class type int - short for integer. An int variable can store a value in the range -32768 to +32767. You can think of it as a largish positive or negative whole number: no fractional part is allowed. To declare an int you use the instruction:

```
int variable name;
```

For example:

```
int a;
```

declares that you want to create an int variable called a.

To assign a value to our integer variable we would use the following C statement:

```
a=10;
```

The C programming language uses the "=" character for assignment. A statement of the form `a=10;` should be interpreted as take the numerical value 10 and store it in a memory location associated with the integer variable a. The "=" character should not be seen as an equality otherwise writing statements of the form:

```
a=a+10;
```

will get mathematicians blowing fuses! This statement should be interpreted as take the current value stored in a memory location associated with the integer variable a; add the numerical value 10 to it and then replace this value in the memory location associated with a.

Decimal Number Variables:

As described above, an integer variable has no fractional part. Integer variables tend to be used for counting, whereas real numbers are used in arithmetic. C uses one of two keywords to declare a variable that is to be associated with a decimal number: `float` and `double`. They each offer a different level of precision as outlined below.

`float`

A float, or floating point, number has about seven digits of precision and a range of about $1.E-36$ to $1.E+36$. A float takes four bytes to store.

`double`

A double, or double precision, number has about 13 digits of precision and a range of about $1.E-303$ to $1.E+303$. A double takes eight bytes to store.

For example:

```
float total;
```

```
double sum;
```

To assign a numerical value to our floating point and double precision variables we would use the following C statement:

```
total=0.0;
```

```
sum=12.50;
```

Character Variables:

C only has a concept of numbers and characters. It very often comes as a surprise to some programmers who learnt a beginner's language such as BASIC that C has no understanding of strings but a string is only an array of characters and C does have a concept of arrays which we shall be meeting later in this course.

To declare a variable of type character we use the keyword `char`. - A single character stored in one byte.

For example:

```
char c;
```

To assign, or store, a character value in a `char` data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if `c` is a `char` variable you can store the letter A in it using the following C statement:

```
c='A'
```

Notice that you can only store a single character in a `char` variable. Later we will be discussing using character strings, which has a very real potential for confusion because a string constant is written between double quotes. But for the moment remember that a `char` variable is 'A' and not "A".

Assignment Statement:

Once you've declared a variable you can use it, but not until it has been declared - attempts to use a variable that

has not been defined will cause a compiler error. Using a variable means storing something in it. You can store a value in a variable using:

```
name = value;
```

For example:

```
a=10;
```

stores the value 10 in the int variable a. What could be simpler? Not much, but it isn't actually very useful! Who wants to store a known value like 10 in a variable so you can use it later? It is 10, always was 10 and always will be 10. What makes variables useful is that you can use them to store the result of some arithmetic.

Consider four very simple mathematical operations: add, subtract, multiply and divide. Let us see how C would use these operations on two float variables a and b.

```
add
  a+b
subtract
  a-b
multiply
  a*b
divide
  a/b
```

Note that we have used the following characters from C's character set:

```
+   for add
-   for subtract
*   for multiply
/   for divide
```

BE CAREFUL WITH ARITHMETIC!!! What is the answer to this simple calculation?

```
a=10/3
```

The answer depends upon how a was declared. If it was declared as type int the answer will be 3; if a is of type float then the answer will be 3.333. It is left as an exercise to the reader to find out the answer for a of type char.

Two points to note from the above calculation:

- 1.C ignores fractions when doing integer division!
- 2.when doing float calculations integers will be converted into float. We will see later how C handles type conversions.

Arithmetic Ordering:

Whilst we are dealing with arithmetic we want to remind you about something that everyone learns at junior school but then we forget it. Consider the following calculation:

```
a=10.0 + 2.0 * 5.0 - 6.0 / 2.0
```

What is the answer? If you think its 27 go to the bottom of the class! Perhaps you got that answer by following each instruction as if it was being typed into a calculator. A computer doesn't work like that and it has its own set of rules when performing an arithmetic calculation. All mathematical operations form a hierachy which is shown here. In the above calculation the multiplication and division parts will be evaluated first and then the addition and subtraction parts. This gives an answer of 17.

Note: To avoid confusion use brackets. The following are two different calculations:

```
a=10.0 + (2.0 * 5.0) - (6.0 / 2.0)
a=(10.0 + 2.0) * (5.0 - 6.0) / 2.0
```

You can freely mix int, float and double variables in expressions. In nearly all cases the lower precision values are

converted to the highest precision values used in the expression. For example, the expression `f*i`, where `f` is a float and `i` is an int, is evaluated by converting the int to a float and then multiplying. The final result is, of course, a float but this may be assigned to another data type and the conversion will be made automatically. If you assign to a lower precision type then the value is truncated and not rounded. In other words, in nearly all cases you can ignore the problems of converting between types.

This is very reasonable but more surprising is the fact that the data type `char` can also be freely mixed with ints, floats and doubles. This will shock any programmer who has used another language, as it's another example of C getting us closer than is customary to the way the machine works. A character is represented as an ASCII or some other code in the range 0 to 255, and if you want you can use this integer code value in arithmetic. Another way of thinking about this is that a `char` variable is just a single-byte integer variable that can hold a number in the range 0 to 255, which can optionally be interpreted as a character. Notice, however, that C gives you access to memory in the smallest chunks your machine works with, i.e. one byte at a time, with no overheads.

Something To Declare:

Before you can use a variable you have to declare it. As we have seen above, to do this you state its type and then give its name. For example, `int i;` declares an integer variable. You can declare any number of variables of the same type with a single statement. For example:

```
int a, b, c;
```

declares three integers: `a`, `b` and `c`. You have to declare all the variables that you want to use at the start of the program. Later you will discover that exactly where you declare a variable makes a difference, but for now you should put variable declarations after the opening curly bracket of the main program.

Here is an example program that includes some of the concepts outlined above. It includes a slightly more advanced use of the `printf` function which will be covered in detail in the next part of this course:

```
/*
/*
   Program#int.c

   Another simple program
   using int and printf
*/

#include

main()
{
    int a,b,average;
    a=10;
    b=6;
    average = ( a+b ) / 2 ;
    printf("Here ");
    printf("is ");
    printf("the ");
    printf("answer... ");
    printf("\n");
    printf("%d.",average);
}
```

More On Initialising Variables:

You can assign an initial value to a variable when you declare it. For example:

```
int i=1;
```

sets the `int` variable to one as soon as it's created. This is just the same as:


```
int i;  
i=1;
```

but the compiler may be able to speed up the operation if you initialise the variable as part of its declaration. Don't assume that an uninitialised variable has a sensible value stored in it. Some C compilers store 0 in newly created numeric variables but nothing in the C language compels them to do so.

Summary:

Variable names:

- should be lowercase for local variables
- should be UPPERCASE for symbolic constants (to be discussed later)
- only the first 31 characters of a variables name are significant
- must begin with a letter or _ (under score) character

Input and Output Functions

Objectives:

Having read this section you should have a clearer idea of one of C's:

1. input functions, called scanf
2. output functions, called printf

On The Run:

Even with arithmetic you can't do very much other than write programs that are the equivalent of a pocket calculator. The real break through comes when you can read values into variables as the program runs. Notice the important words here: "as the program runs". You can already store values in variables using assignment. That is:

```
a=100;
```

stores 100 in the variable a each time you run the program, no matter what you do. Without some sort of input command every program would produce exactly the same result every time it was run. This would certainly make debugging easy! But in practice, of course, we need programs to do different jobs each time they are run. There are a number of different C input commands, the most useful of which is the scanf command. To read a single integer value into the variable called a you would use:

```
scanf("%d",&a);
```

For the moment don't worry about what the %d or the >&a means - concentrate on the difference between this and:

```
a=100;
```

When the program reaches the scanf statement it pauses to give the user time to type something on the keyboard and continues only when users press <Enter>, or <Return>, to signal that he, or she, has finished entering the value. Then the program continues with the new value stored in a. In this way, each time the program is run the user gets a chance to type in a different value to the variable and the program also gets the chance to produce a different result!

The final missing piece in the jigsaw is using the printf function, the one we have already used to print "Hello World", to print the value currently being stored in a variable. To display the value stored in the variable a you would use:

```
printf("The value stored in a is %d",a);
```

The %d, both in the case of scanf and printf, simply lets the compiler know that the value being read in, or printed out, is a decimal integer - that is, a few digits but no decimal point.

Note: the scanf function does not prompt for an input. You should get in the habit of always using a printf function, informing the user of the program what they should type, before a scanf function.

Input and Output Functions in More Detail:

One of the advantages of C is that essentially it is a small language. This means that you can write a complete description of the language in a few pages. It doesn't have many keywords or data types for that matter. What makes C so powerful is the way that these low-level facilities can be put together to make higher level facilities.

The only problem with this is that C programmers have a tendency to reinvent the wheel each time they want to go for a ride. It is also possible to write C programs in a variety of styles which depend on the particular tricks and devices that a programmer chooses to use. Even after writing C for a long time you will still find the occasionally construction which makes you think, "I never thought of that!" or, "what is that doing?"

One attempt to make C a more uniform language is the provision of standard libraries of functions that perform

common tasks. We say standard but until the ANSI committee actually produced a standard there was, and still is, some variation in what the standard libraries contained and exactly how the functions worked. Having said that we had better rush in quickly with the reassurance that in practice the situation isn't that bad and most of the functions that are used frequently really are standard on all implementations. In particular the I/O functions vary very little.

It is now time to look at exactly how scanf and printf work and what they can do - you might be surprised at just how complex they really are!

The original C specification did not include commands for input and output. Instead the compiler writers were supposed to implement library functions to suit their machines. In practice all chose to implement printf and scanf and after a while C programmers started to think of them as if these functions were I/O keywords! It sometimes helps to remember that they are functions on a par with any other functions you may care to define. If you want to you can provide your own implementations of printf or scanf or any of the other standard functions - we'll discover how later.

printf:

The printf (and scanf) functions do differ from the sort of functions that you will create for yourself in that they can take a variable number of parameters. In the case of printf the first parameter is always a string (c.f. "Hello World") but after that you can include as many parameters of any type that you want to. That is, the printf function is usually of the form:

```
printf(string,variable,variable,variable...)
```

where the ... means you can carry on writing a list of variables separated by commas as long as you want to. The string is all-important because it specifies the type of each variable in the list and how you want it printed. The string is usually called the control string or the format string. The way that this works is that printf scans the string from left to right and prints on the screen, or any suitable output device, any characters it encounters - except when it reaches a % character. The % character is a signal that what follows it is a specification for how the next variable in the list of variables should be printed. printf uses this information to convert and format the value that was passed to the function by the variable and then moves on to process the rest of the control string and any more variables it might specify. For example:

```
printf("Hello World");
```

only has a control string and, as this contains no % characters it results in Hello World being displayed and doesn't need to display any variable values. The specifier %d means convert the next value to a signed decimal integer and so:

```
printf("Total = %d",total);
```

will print Total = and then the value passed by >total as a decimal integer.

If you are familiar with other programming languages then you may feel happy about the printf function because something like:

```
printf("Total = %d",total);
```

looks like the sort of output command you might have used before. For example, in BASIC you would write:

```
PRINT "Total = ",total
```

but the C view of output is at a lower level than you might expect. The %d isn't just a format specifier, it is a conversion specifier. It indicates the data type of the variable to be printed and how that data type should be converted to the characters that appear on the screen. That is %d says that the next value to be printed is a signed integer value (i.e. a value that would be stored in a standard int variable) and this should be converted into a sequence of characters (i.e. digits) representing the value in decimal. If by some accident the variable that you are trying to display happens to be a float or a double then you will still see a value displayed - but it will not correspond to the actual value of the float or double.

The reason for this is twofold.

1. The first difference is that an int uses two bytes to store its value, while a float uses four and a double uses

eight. If you try to display a float or a double using %d then only the first two bytes of the value are actually used.

2. The second problem is that even if there wasn't a size difference ints, floats and doubles use a different binary representation and %d expects the bit pattern to be a simple signed binary integer.

This is all a bit technical, but that's in the nature of C. You can ignore these details as long as you remember two important facts:

1. The specifier following % indicates the type of variable to be displayed as well as the format in which that the value should be displayed;

2. If you use a specifier with the wrong type of variable then you will see some strange things on the screen and the error often propagates to other items in the printf list.

If this seems complicated then I would agree but I should also point out that the benefit is being able to treat what is stored in a variable in a more flexible way than other languages allow.

Other languages never let on to the programmer that what is in fact stored in a variable is a bit pattern, not the decimal value that appears to be stored there when you use a printf (or whatever) statement. Of course whether you view this as an advantage depends on what you are trying to do. It certainly brings you closer to the way the machine works.

You can also add an 'l' in front of a specifier to mean a long form of the variable type and h to indicate a short form (long and short will be covered later in this course). For example, %ld means a long integer variable (usually four bytes) and %hd means short int. Notice that there is no distinction between a four-byte float and an eight-byte double. The reason is that a float is automatically converted to a double precision value when passed to printf - so the two can be treated in the same way. (In pre-ANSI all floats were converted to double when passed to a function but this is no longer true.) The only real problem that this poses is how to print the value of a pointer? The answer is that you can use %x to see the address in hex or %o to see the address in octal. Notice that the value printed is the segment offset and not the absolute address - to understand what we are going on about you need to know something about the structure of your processor.

The % Format Specifiers:

The % specifiers that you can use in ANSI C are:

	Usual variable type	Display
%c	char	single character
%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%f	float or double	signed decimal
%g (%G)	float or double	use %f or %e as required
%o	int	unsigned octal value
%p	pointer	address stored in pointer
%s	array of char	sequence of characters
%u	int	unsigned decimal
%x (%X)	int	unsigned hex value

Formatting Your Output:

The type conversion specifier only does what you ask of it - it convert a given bit pattern into a sequence of characters that a human can read. If you want to format the characters then you need to know a little more about the printf function's control string.

Each specifier can be preceded by a modifier which determines how the value will be printed. The most general modifier is of the form:

flag width.precision

The flag can be any of:

flag	meaning
-	left justify
+	always display sign

space	display space if there is no sign
0	pad with leading zeros
#	use alternate form of specifier

The width specifies the number of characters used in total to display the value and precision indicates the number of characters used after the decimal point.

For example, %10.3f will display the float using ten characters with three digits after the decimal point. Notice that the ten characters includes the decimal point, and a - sign if there is one. If the value needs more space than the width specifies then the additional space is used - width specifies the smallest space that will be used to display the value. (This is quiet reassuring, you won't be the first programmer whose program takes hours to run but the output results can't be viewed because the wrong format width has been specified!)

The specifier %-10d will display an int left justified in a ten character space. The specifier %+5d will display an int using the next five character locations and will add a + or - sign to the value.

The only complexity is the use of the # modifier. What this does depends on which type of format it is used with:

%#o	adds a leading 0 to the octal value
%#x	adds a leading 0x to the hex value
%#f or	
%#e	ensures decimal point is printed
%#g	displays trailing zeros

Strings will be discussed later but for now remember: if you print a string using the %s specifier then all of the characters stored in the array up to the first null will be printed. If you use a width specifier then the string will be right justified within the space. If you include a precision specifier then only that number of characters will be printed.

For example:

```
printf("%s,Hello")
```

will print Hello,

```
printf("%25s ,Hello")
```

will print 25 characters with Hello right justified and

```
printf("%25.3s,Hello")
```

will print Hello right justified in a group of 25 spaces.

Also notice that it is fine to pass a constant value to printf as in printf("%s,Hello").

Finally there are the control codes:

\b	backspace
\f	formfeed
\n	new line
\r	carriage return
\t	horizontal tab
\'	single quote
\0	null

If you include any of these in the control string then the corresponding ASCII control code is sent to the screen, or output device, which should produce the effect listed. In most cases you only need to remember \n for new line.

scanf:

Now that we have mastered the intricacies of printf you should find scanf very easy. The scanf function works in

much the same way as the printf. That is it has the general form:

```
scanf(control string,variable,variable,...)
```

In this case the control string specifies how strings of characters, usually typed on the keyboard, should be converted into values and stored in the listed variables. However there are a number of important differences as well as similarities between scanf and printf.

The most obvious is that scanf has to change the values stored in the parts of computers memory that is associated with parameters (variables).

To understand this fully you will have to wait until we have covered functions in more detail. But, just for now, bare with us when we say to do this the scanf function has to have the addresses of the variables rather than just their values. This means that simple variables have to be passed with a preceding >&. (Note for future reference: There is no need to do this for strings stored in arrays because the array name is already a pointer.)

The second difference is that the control string has some extra items to cope with the problems of reading data in. However, all of the conversion specifiers listed in connection with printf can be used with scanf.

The rule is that scanf processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value. If you input multiple values then these are assumed to be separated by white space - i.e. spaces, newline or tabs. This means you can type:

```
3 4 5
```

or

```
3
4
5
```

and it doesn't matter how many spaces are included between items. For example:

```
scanf("%d %d",&i,&j);
```

will read in two integer values into i and j. The integer values can be typed on the same line or on different lines as long as there is at least one white space character between them.

The only exception to this rule is the %c specifier which always reads in the next character typed no matter what it is. You can also use a width modifier in scanf. In this case its effect is to limit the number of characters accepted to the width.

For example:

```
scanf("%lOd",&i)
```

would use at most the first ten digits typed as the new value for i.

There is one main problem with scanf function which can make it unreliable in certain cases. The reason being is that scanf tends to ignore white spaces, i.e. the space character. If you require your input to contain spaces this can cause a problem. Therefore for string data input the function getstr() may well be more reliable as it records spaces in the input text and treats them as an ordinary characters.

Custom Libraries:

If you think printf and scanf don't seem enough to do the sort of job that any modern programmer expects to do, you would be right. In the early days being able to print a line at a time was fine but today we expect to be able to print anywhere on the screen at any time.

The point is that as far as standard C goes simple I/O devices are stream-oriented - that is you send or get a stream of characters without any notion of being able to move the current position in the stream. If you want to move backwards and forwards through the data then you need to use a direct access file. In more simple terms, C doesn't have a Tab(X,Y) or

Locate(X,Y) function or command which moves the cursor to the specified location! How are you ever going to write your latest block buster game, let alone build your sophisticated input screens?

Well you don't have to worry too much because although C may not define them as standard, all C implementations come with an extensive graphics/text function library that allows you to do all of this and more. Such a library isn't standard, however the principles are always the same. The Borland and Microsoft offerings are usually considered as the two facto standards.

Summing It Up:

Now that we have arithmetic, a way of reading values in and a way of displaying them, it's possible to write a slightly more interesting program than "Hello World". Not much more interesting, it's true, but what do you expect with two instructions and some arithmetic?

Let's write a program that adds two numbers together and prints the result. (I told you it wasn't that much more interesting!) Of course, if you want to work out something else like fahrenheit to centigrade, inches to centimetres or the size of your bank balance, then that's up to you - the principle is the same.

The program is a bit more complicated than you might expect, but only because of the need to let the user know what is happening:

```
#include <stdio.h>
main()
{
    int a,b,c;
    printf("\nThe first number is ");
    scanf("%d",&a);
    printf("The second number is ");
    scanf("%d",&b);
    c=a+b;
    printf("The answer is %d \n",c);
}
```

The first instruction declares three integer variables: a, b and c. The first two printf statements simply display message on the screen asking the user for the values. The scanf functions then read in the values from the keyboard into a and b. These are added together and the result in c is displayed on the screen with a suitable message. Notice the way that you can include a message in the printf statement along with the value.

Type the program in, compile it and link it and the result should be your first interactive program. Try changing it so that it works out something a little more adventurous. Try changing the messages as well. All you have to remember is that you cannot store values or work out results greater than the range of an integer variable or with a fractional part.

Control Loops

Objectives:

Having read this section you should have an idea about C's:

1. Conditional, or Logical, Expressions as used in program control
2. the do while loop
3. the while loop
4. the for loop

Go With The Flow:

Our programs are getting a bit more sophisticated, but they still lack that essential something that makes a computer so necessary. Exactly what they lack is the most difficult part to describe to a beginner. There are only two great ideas in computing. The first is the variable and you've already met that. The second is flow of control.

When you write a list of instructions for someone to perform you usually expect them to follow the list from the top to the bottom, one at a time. This is the simple default flow of control through a program. The C programs we have written so far use this one-after-another default flow of control.

This is fine and simple, but it limits the running time of any program we can write. Why? Simply because there is a limit to the number of instructions you can write and it doesn't take long for a computer to read though and obey your list. So how is it that we have programs that run for hours on end if need be? The answer is statements that alter the one-after-another order of obeying instructions. Perhaps the most useful is the loop.

Suppose we ask you to display "Hello World!" five times on the screen. Easy! you'd say:

```
#include <stdio.h>
main()
{
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
}
```

Indeed, this does exactly what was asked. But now we up the bet and ask you to do the same job for 100 hellos or, if you're still willing to type that much code, maybe 1,000 Hello World's, 10,000 Hello World's, or whatever it takes you to realise this isn't a sensible method!

What you really need is some way of repeating the printf statements without having to write it out each time. The solution to this problem is the while loop or the do while loop.

The while and do while Loops:

You can repeat any statement using either the while loop:

```
while(condition) compound statement;
```

or the do while loop:

```
do compound statement while(condition);
```

The condition is just a test to control how long you want the compound statement to carry on repeating.

Each line of a C program up to the semicolon is called a statement. The semicolon is the statement's terminator. The braces { and } which have appeared at the beginning and end of our program unit can also be used to group together related declarations and statements into a compound statement or a block.

In the case of the while loop before the compound statement is carried out the condition is checked, and if it is true the statement is obeyed one more time. If the condition turns out to be false, the looping isn't obeyed and the program moves on to the next statement. So you can see that the instruction really means while something or other is true keep on doing the statement.

In the case of the do while loop it will always execute the code within the loop at least once, since the condition controlling the loop is tested at the bottom of the loop. The do while loop repeats the instruction while the condition is true. If the condition turns out to be false, the looping isn't obeyed and the program moves on to the next statement.

Conditions or Logical Expressions:

The only detail we need to clear up is what the condition (or Logical Expression) can be. How, for example, do we display 100 or 10,000 "Hello World!" messages? The condition can be any test of one value against another. For example:

```
a>0
```

is true if a contains a value greater than zero;

```
b<0
```

is true if b contains a value less than zero.

The only complication is that the test for 'something equals something else' uses the character sequence == and not =. That's right: a test for equality uses two equal-signs, as in a==0, while an assignment, as in a=0, uses one. This use of the double or single equal sign to mean slightly different things is a cause of many a program bug for beginner and expert alike!

So what about answering the question? What about the 100 "Hello World"s? Well, for the moment we know easily how to produce an infinite number of Hello World!"s using while loop:

```
#include <stdio.h>
main()
{
    while (1 == 1) printf("Hello World!\n");
}
```

and using the do while loop:

```
#include <stdio.h>
main()
{
    do
        printf("Hello World!\n");
    while (1 == 1)
}
```

If you type either of these programs in and run it you will find that your screen fills with a never ending list of "Hello World!"s. Why? Because the condition to keep the repeat going is (1 == 1), one equals one in plain English, which is always true! So how do we stop the loop? In some cases it could be by pulling the plug out - but usually you can stop an infinite loop by pressing Ctrl-Break or Ctrl-C.

An infinite loop is sometimes useful - I certainly hope the program controlling the nearest nuclear power station is an infinite loop that never receives a Ctrl-Break signal! Most loops, however, have to stop some time.

To solve our problem of printing 100 "Hello World!"s we need a counter and a test for when that counter reaches 100. A counter is a simple variable that has one added to it each time through the loop, using an instruction like this:

```
a=a+1;
```

This always confuses beginners, because they aren't used to seeing the variable on both sides of the equal-sign. All this means is that a has one added to it to produce a new value, and this value is stored back in the location called a. If you're worried, try thinking about it as:

```
temp = a+1;
a    = temp;
```

The two approaches are more or less the same. C is a language where anything that's used often can be said concisely, so it lets you say "add one to a variable" using the shorter notation:

```
++a;
```

The double plus is read "increment a by one". Make sure you know that ++a; and a=a+1; are the same thing because you will often see both in typical C programs.

The increment operator ++ and the equivalent decrement operator --, can be used as either prefix (before the variable) or postfix (after the variable). Note: ++a increments a before using its value; whereas a++ which means use the value in a then increment the value stored in a.

Now it is easy to print "Hello World!" 100 times using the while loop:

```
#include <stdio.h>
main()
{
    int count;
    count=0;
    while (count < 100)
    {
        ++count;
        printf("Hello World!\n");
    }
}
```

[program]

or the do while loop:

```
#include <stdio.h>
main()
{
    int count;
    count=0;
    do
    {
        ++count;
        printf("Hello, World!\n");
    } while (count < 100)
}
```

Note: the use of the { and } to form a compound statement; all statements between the braces will be executed before the loop check is made.

The integer variable count is declared and then set to zero, ready to count the number of times we have gone round the loop. Each time round the loop the value of count is checked against 100. As long as it is less, the loop carries on. Each time the loop carries on, count is incremented and "Hello World!" is printed - so eventually count does reach 100 and the loop stops.

These little programs are just a bit more subtle than you might think. Ask yourself, do they really print exactly 100 times? Ask yourself: what is the final value of count? If you want to make sure you are right change the printf to:

```
printf("count is %d",count);
```

and add a printf after the loop:

```
printf("final value is %d",count);
```

Make sure you understand why you get the results that you do. What would happen if you changed the initial value of count to be one rather than zero?

Looping the Loop:

We have seen that any list of statements enclosed in curly brackets is treated as a single statement, a compound statement. So to repeat a list of statements all you have to do is put them inside a pair of curly brackets as in:

```
while (condition)
{
    statement1;
    statement2;
    statement3;
}
```

which repeats the list while the condition is true. Notice that the statements within the curly brackets have to be terminated by semicolons as usual. Notice also that as the while statement is a complete statement it too has to be terminated by a semi-colon - except for the influence of one other punctuation rule. You never have to follow a right curly bracket with a semi-colon. This rule was introduced to make C look tidier by avoiding things like

```
};};};}
```

at the end of a complicated program. You can write the semi-colon after the right bracket if you want to, but most C programmers don't. You can use a compound statement anywhere you can use a single statement.

The for Loop:

The while, and do-while, loop is a completely general way of repeating a section of program over and over again - and you don't really need anything else but... The while loop repeats a list of instructions while some condition or other is true and often you want to repeat something a given number of times.

The traditional solution to this problem is to introduce a variable that is used to count the number of times that a loop has been repeated and use its value in the condition to end the loop. For example, the loop:

```
i=1;
while (i<10)
{
    printf("%d \n",i);
    ++i;
}
```

repeats while i is less than 10. As the ++ operator is used to add one to i each time through the loop you can see that i is a loop counter and eventually it will get bigger than 10, i.e. the loop will end.

The question is how many times does the loop go round? More specifically what values of i is the loop carried out for? If you run this program snippet you will find that it prints 1,2,3... and finishes at 10. That is, the loop repeats 10 times for values of i from 1 to 10. This sort of loop - one that runs from a starting value to a finishing value going up by one each time - is so common that nearly all programming languages provide special commands to implement it. In C this special type of loop can be implemented as a for loop.

```
for ( counter=start_value; counter <= finish_value; ++counter )
    compound statement
```

which is entirely equivalent to:

```

counter=start;
while (couner <= finish)
{
    statements;
    ++counter;
}

```

The condition operator <= should be interpreted as less than or equal too. We will be covering all of C's conditions , or logical expressions, in the next section.

For example to print the numbers 1 to 100 you could use:

```
for ( i=1; i <= 100; ++i ) printf("%d \n",i);
```

You can, of course repeat a longer list of instructions simply by using a compound statement.

The C for loop is much more flexible than this simple description. Indeed, many would be horrified at the way we have described the for loop without displaying its true generality, but keep in mind that there is more to come.

In the meantime consider the following program, it does a temperature conversion, but it also introduces one or two new concepts:

- 1.our counter does not have to be incremented (deremented) by 1; we can use any value.
- 2.we can do calculations within the printf statement.

```

#include <stdio.h>

main()
{
    int fahr;

    for ( fahr = 0 ; fahr <= 300 ; fahr = fahr + 20)
        printf("%4d %6.1f\n" , fahr , (5.0/9.0)*(fahr-32));
}

```

and here's another one for you to look at:

```

#include <stdio.h>

main()
{
    int lower , upper , step;
    float fahr , celsius;

    lower = 0 ;
    upper = 300;
    step = 20 ;

    fahr = lower;

    while ( fahr <= upper ) {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf("%4.0f %6.1f\n" , fahr , celsius);
        fahr = fahr + step;
    }
}

```

Conditional Execution

Objectives:

Having read this section you should be able to:

1. Program control with if, if-else and switch structures
2. have a better idea of what C understands as true and false.

Program Control:

It is time to turn our attention to a different problem - conditional execution. We often need to be able to choose which set of instructions are obeyed according to a condition. For example, if you're keeping a total and you need to display the message 'OK' if the value is greater than zero you would need to write something like:

```
if (total>0) printf("OK");
```

This is perfectly reasonable English, if somewhat terse, but it is also perfectly good C. The if statement allows you to evaluate a >condition and only carry out the statement, or compound statement, that follows if the condition is true. In other words the printf will only be obeyed if the condition total > 0 is true.

If the condition is false then the program continues with the next instruction. In general the if statement is of the following form:

```
if (condition) statement;
```

and of course the statement can be a compound statement.

Here's an example program using two if statements:

```
#include <stdio.h>

main()
{
    int a , b;

    do {

        printf("\nEnter first number: ");
        scanf("%d" , &a);

        printf("\nEnter second number: ");
        scanf("%d" , &b);

        if (a<b) printf("\n\nFirst number is less than second\n\n");
        if (b<a) printf("Second number is less than first\n\n");

    } while (a < 999);
}
```

Here's another program using an if keyword and a compound statement or a block:

```
#include <stdio.h>

main()
{
    int a , b;

    do {

        printf("\nEnter first number: ");
        scanf("%d" , &a);
```

```

printf("\nEnter second number: ");
scanf("%d" , &b);

if (a<b) {
    printf("\n\nFirst number is less than second\n");
    printf("Their difference is : %d\n" , b-a);
    printf("\n");
}

printf("\n");

} while (a < 999);
}

```

The if statement lets you execute or skip an instruction depending on the value of the condition. Another possibility is that you might want to select one of two possible statements - one to be obeyed when the condition is true and one to be obeyed when the condition is false. You can do this using the

```

if (condition) statement1;
else statement2;

```

form of the if statement.

In this case statement1 is carried out if the condition is true and statement2 if the condition is false.

Notice that it is certain that one of the two statements will be obeyed because the condition has to be either true or false! You may be puzzled by the semicolon at the end of the if part of the statement. The if (condition) statement1 part is one statement and the else statement2 part behaves like a second separate statement, so there has to be semi-colon terminating the first statement.

Logical Expressions:

So far we have assumed that the way to write the conditions used in loops and if statements is so obvious that we don't need to look more closely. In fact there are a number of deviations from what you might expect. To compare two values you can use the standard symbols:

```

> (greater than)
< (less than)
>= (for greater than or equal to )
<= (for less than or equal to)
== (to test for equality).

```

The reason for using two equal signs for equality is that the single equals sign always means store a value in a variable - i.e. it is the assignment operator. This causes beginners lots of problems because they tend to write:

```

if (a = 10) instead of if (a == 10)

```

The situation is made worse by the fact that the statement if (a = 10) is legal and causes no compiler error messages! It may even appear to work at first because, due to a logical quirk of C, the assignment actually evaluates to the value being assigned and a non-zero value is treated as true (see below). Confused? I agree it is confusing, but it gets easier. . .

Just as the equals condition is written differently from what you might expect so the non-equals sign looks a little odd. You write not equals as !=. For example:

```

if (a != 0)

```

is 'if a is not equal to zero'.

An example program showing the if else construction now follows:

```

#include <stdio.h>

main ()
{
    int num1, num2;

    printf("\nEnter first number ");
    scanf("%d",&num1);

    printf("\nEnter second number ");
    scanf("%d",&num2);

    if (num2 ==0) printf("\n\nCannot devide by zero\n\n");
    else          printf("\n\nAnswer is %d\n\n",num1/num2);
}

```

This program uses an if and else statement to prevent division by 0 from occurring.

True and False in C:

Now we come to an advanced trick which you do need to know about, but if it only confuses you, come back to this bit later. Most experienced C programmers would wince at the expression `if(a!=0)`.

The reason is that in the C programming language doesn't have a concept of a Boolean variable, i.e. a type class that can be either true or false. Why bother when we can use numerical values. In C true is represented by any numeric value not equal to 0 and false is represented by 0. This fact is usually well hidden and can be ignored, but it does allow you to write

`if(a != 0)` just as `if(a)`

because if a isn't zero then this also acts as the value true. It is debatable if this sort of shortcut is worth the three characters it saves. Reading something like

`if(!done)`

as 'if not done' is clear, but `if(!total)` is more dubious.

Using break and continue Within Loops:

The break statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. The break statement can be used with all three of C's loops. You can have as many statements within a loop as you desire. It is generally best to use the break for special purposes, not as your normal loop exit. break is also used in conjunction with functions and >case statements which will be covered in later sections.

The continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. In while and do-while loops, a continue statement will cause control to go directly to the test condition and then continue the looping process. In the case of the for loop, the increment part of the loop continues. One good use of continue is to restart a statement sequence when an error occurs.

```

#include <stdio.h>

main()
{
    int x ;

    for ( x=0 ; x<=100 ; x++) {
        if (x%2) continue;
        printf("%d\n" , x);
    }
}

```

Here we have used C's modulus operator: %. A expression:

`a % b`

produces the remainder when a is divided by b; and zero when there is no remainder.

Here's an example of a use for the break statement:

```
#include <stdio.h>

main()
{
    int t ;

    for ( ; ; ) {
        scanf("%d" , &t) ;
        if ( t==10 ) break ;
    }
    printf("End of an infinite loop...\n");
}
```

Select Paths with switch:

While if is good for choosing between two alternatives, it quickly becomes cumbersome when several alternatives are needed. C's solution to this problem is the switch statement. The switch statement is C's multiple selection statement. It is used to select one of several alternative paths in program execution and works like this: A variable is successively tested against a list of integer or character constants. When a match is found, the statement sequence associated with the match is executed. The general form of the switch statement is:

```
switch(expression)
{
    case constant1:  statement sequence; break;
    case constant2:  statement sequence; break;
    case constant3:  statement sequence; break;
    .
    .
    .
    default:  statement sequence; break;
}
```

Each case is labelled by one, or more, constant expressions (or integer-valued constants). The default statement sequence is performed if no matches are found. The default is optional. If all matches fail and default is absent, no action takes place.

When a match is found, the statement sequence associated with that case are executed until break is encountered.

An example program follows:

```
#include <stdio.h>

main()
{
    int i;

    printf("Enter a number between 1 and 4");
    scanf("%d",&i);

    switch (i)
    {
        case 1:
```



```
    printf("one");
    break;
case 2:
    printf("two");
    break;
case 3:
    printf("three");
    break;
case 4:
    printf("four");
    break;
default:
    printf("unrecognized number");
} /* end of switch */

}
```

This simple program recognizes the numbers 1 to 4 and prints the name of the one you enter. The switch statement differs from if, in that switch can only test for equality, whereas the if conditional expression can be of any type. Also switch will work with only int and char types. You cannot for example, use floating-point numbers. If the statement sequence includes more than one statement they will have to be enclosed with { } to form a compound statement.

Structure and Nesting

Objectives:

This section brings together the various looping mechanisms available to the C programmer with the program control constructs we met in the last section.

We also demonstrates a neat trick with random numbers.

It is one of the great discoveries of programming that you can write any program using just simple while loops and if statements. You don't need any other control statements at all. Of course it might be nice to include some other types of control statement to make life easy - for example, you don't need the for loop, but it is good to have! So as long as you understand the if and the while loop in one form or another you can write any program you want to.

If you think that a loop and an if statement are not much to build programs then you are missing an important point. It's not just the statements you have, but the way you can put them together. You can include an if statement within a loop, loops within loops are also OK, as are loops in ifs, and ifs in ifs and so on. This putting one control statement inside another is called nesting and it is really what allows you to make a program as complicated as you like.

Think of a number:

Now let's have a go at writing the following program: 'It thinks of a number in the range 0 to 99 and then asks the user to guess it'. This sounds complicated, especially the 'thinks of a number' part, but all you need to know is that the statement:

```
r = rand()
```

will store a random number in the integer variable r. The standard library function rand() randomly picks a number within the range 0 to 32767, but this might vary from machine to machine. Look upon rand() as being a large dice.

Our problem is to select a number between 0 and 99 and not between 0 and 32767. How can we get our random number to within our range? The rand() function will produce numbers such as:

```
2567
134
20678
15789
32001
15987
etc...
```

If you look at the last two digits of all of these numbers they would form our random set! To select just these numbers we can use an arithmetic calculation of the following form:

```
r = rand() % 100
```

That is, to get the number into the right range you simply take the remainder on dividing by 100, ie a value in the range 0 to 99. You should remember this neat programming trick, you'll be surprised how often it is required.

Our solution to the problem is as follows:

```
#include <stdio.h>

main()
{
    int target;
    int guess;
    int again;
```

```

printf("\n Do you want to guess a number 1 =Yes, 0=No ");
scanf("%d",&again);

while (again)
{
    target = rand() % 100;
    guess  = target + 1;

    while(target!=guess)
    {
        printf("\n What is your guess ? ");
        scanf("%d",&guess);

        if (target>guess) printf("Too low");
        else printf("Too high");
    }

    printf("\n Well done you got it! \n");
    printf("\nDo you want to guess a number 1=Yes, 0=No");
    scanf("%d",&again);
}
}

```

This looks like a very long and complicated program, but it isn't. Essentially it used two loops and an if/else which in English could be summarised as:

```

while(again) {
    think of a number
    while (user hasn't guessed it)
    {
        get users guess.
        if (target < guess) tell the user the guess is low
        else                tell the user the guess is high
    }
}

```

The integer variable again is used to indicate that the user wants to carry on playing. If it is 0 then the loop stops so 0 = No, and 1, or any other non-zero value, = Yes.

If you try this program out you will discover that it has a slight flaw - not so much a bug, more a feature. If the user guesses the correct value the program still tells the user that the guess is too high and then congratulates them that they have the correct value. Such problems with how loops end are common and you have to pay attention to details such as this. There are a number of possible solutions, but the most straight forward is to change the inner loop so that the first guess is asked for before the loop begins. This shifts the test for the loop to stop to before the test for a high or low guess:

```

#include <stdio.h>

main()
{
    int target;
    int guess;
    int again;

    printf("\n Do you want to guess a number 1 =Yes, 0=No ");
    scanf("%d",&again);

    while (again)
    {
        target = rand() % 100;

        printf("\n What is your guess ? ");
        scanf("%d",&guess);

        while(target!=guess)

```

```
    {
      if (target>guess) printf("Too low");
      else printf("Too high");
      printf("\n What is your guess ? ");
      scanf("%d",&guess);
    }

    printf("\n Well done you got it! \n");
    printf("\n Do you want to guess a number 1=Yes, 0=No");
    scanf("%d",&again);
  }
}
```

If you want to be sure that you understand what is going on here, ask yourself why the line:

```
guess = target + 1;
```

was necessary in the first version of the program and not in the second?

Functions and Prototypes

Objectives:

Having read this section you should be able to:

1. program using correctly defined C functions
2. pass the value of local variables into your C functions

Functions - C's Building Blocks:

Some programmers might consider it a bit early to introduce the C function - but we think you can't get to it soon enough. It isn't a difficult idea and it is incredibly useful. You could say that you only really start to find out what C programming is all about when you start using functions.

C functions are the equivalent of what in other languages would be called subroutines or procedures. If you are familiar with another language you also need to know that C only has functions, so don't spend time looking for the definition of subroutines or procedures - in C the function does everything!

A function is simply a chunk of C code (statements) that you have grouped together and given a name. The value of doing this is that you can use that "chunk" of code repeatedly simply by writing its name. For example, if you want to create a function that prints the word "Hello" on the screen and adds one to variable called total then the chunk of C code that you want to turn into a function is just:

```
printf("Hello");
total = total + 1;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
demo()
{
    printf("Hello");
    total = total + 1;
}
```

Don't worry for now about the curved brackets after the function's name. Once you have defined your function you can use it within a program:

```
main()
{
    demo();
}
```

In this program the instruction `demo ()`; is entirely equivalent to writing out all of the statements in the function. What we have done is to create a new C function and this, of course, is the power of functions. When you are first introduced to the idea of functions, or their equivalent in other languages, it is easy to fall into the trap of thinking that they are only useful when you want to use a block of code more than once.

Functions are useful here but they have a more important purpose. If you are creating a long program then functions allow you to split it into "bite-sized" chunks which you can work on in isolation. As every C programmer knows, "functions are the building blocks of programs."

Functions and Local Variables:

Now that the philosophy session is over we have to return to the details - because as it stands the demo function

will not work. The problem is that the variable total isn't declared anywhere. A function is a complete program sub-unit in its own right and you can declare variables within it just as you can within the main program. If you look at the main program we have been using you will notice it is in fact a function that just happens to be called "main"! So to make demo work we have to add the declaration of the variable total:

```
demo()
{
    int total;
    printf("Hello");
    total=total+1;
}
```

Now this raises the question of where exactly total is a valid variable. You can certainly use total within the function that declares it - this much seems reasonable - but what about other functions and, in particular, what about the main program? The simple answer is that total is a variable that belongs to the demo function. It cannot be used in other functions, it doesn't even exist in other functions and it certainly has nothing to do with any variable of the same name that you declare within other functions.

This is what we hinted at when we said that functions were isolated chunks of code. Their isolation is such that variables declared within the function can only be used within that function. These variables are known as local variables and as their name suggests are local to the function they have been declared in. If you are used to a language where every variable is usable all the time this might seem silly and restrictive - but it isn't. It's what makes it possible to break a large program down into smaller and more manageable chunks.

The fact that total is only usable within the demo function is one thing - but notice we said that it only existed within this function, which is a more subtle point. The variables that a function declares are created when the function is started and destroyed when the function is finished. So if the intention is to use total to count the number of times the >demo function is used - forget it! Each time demo is used the variable total is created afresh, and at the end of the function the variable goes up in a puff of smoke along with its value. So no matter how many times you run demo total will only ever reach a value of 1, assuming that it's initialised to 0.

Making The Connections:

Functions are isolated, and what's more nothing survives after they have finished. Put like this a function doesn't seem to be that useful because you can't get data values in, you can't get data values out, and they don't remember anything that happens to them!

To be useful there has to be a way of getting data into and out of a function, and this is the role of the curved brackets. You can define special variables called parameters which are used to carry data values into a function. Parameters are listed and declared in between the () brackets in the function's definition. For example:

```
sum( int a, int b)
{
    int result;
    result=a + b;
}
```

defines a function called sum with two parameters a and b, both integers. Notice that the result variable is declared in the usual way within the body of the function. Also, notice that the parameters a and b are used within the function in the same way as normal variables - which indeed they are. What is more, they are still local variables and have nothing at all to do with any variables called a and b defined in any other function.

The only way in which parameters are any different is that you can give them initial values when the function starts by writing the values between the round brackets. So

```
sum(1,2);
```

is a call to the sum function with a set to 1 and b set to 2 and so result is set to 3. You can also initialise parameters

to the result of expressions such as:

```
sum(x+2,z*10);
```

which will set a equal to whatever x+2 works out to be and b equal to whatever z*10 works out to be.

As a simpler case you can also set a parameter to the value in a single variable - for example:

```
sum(x,y);
```

will set a to the value stored in x and b to the value stored in y.

Parameters are the main way of getting values into a function, but how do we get values out? There is no point in expecting the >result variable to somehow magically get its value out of the sum function - after all, it is a local variable and is destroyed when sum is finished. You might try something like:

```
sum(int a, int b, int result)
{
    int result;
    result = a + b;
}
```

but it doesn't work. Parameters are just ordinary variables that are set to an initial value when the function starts running - they don't pass values back to the program that used the function. That is:

```
sum(1,2,r);
```

doesn't store 1+2 in r because the value in r is used to initialise the value in result and not vice versa. You can even try

```
sum(1,2,result);
```

and it still will not work - the variable result within the function has nothing to do with the variable result used in any other program.

The simplest way to get a value out of a function is to use the return instruction. A function can return a value via its name - it's as if the name was a variable and had a value. The value that is returned is specified by the instruction:

```
return value;
```

which can occur anywhere within the function, not just as the last instruction - however, a return always terminates the function and returns control back to the calling function. The only complication is that as the function's name is used to return the value it has to be given a data type. This is achieved by writing the data type in front of the function's name. For example:

```
int sum(a,b);
```

So now we can at last write the correct version of the sum function:

```
int sum(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

and to use it you would write something like:

```
r=sum(1,2);
```

which would add 1 to 2 and store the result in r. You can use a function anywhere that you can use a variable. For example,

```
r=sum(1,2)*3
```

is perfectly OK, as is

```
r=3+sum(1,2)/n-10
```

Obviously, the situation with respect to the number of inputs and outputs of a function isn't equal. That is you can create as many parameters as you like but a function can return only a single value. (Later on we will have to find ways of allowing functions to return more than one value.)

So to summarise: a function has the general form:

```
type FunctionName(type declared parameter list)
{
    statements that make up the function
}
```

and of course a function can contain any number of return statements to specify its return value and bring the function to an end.

There are some special cases and defaults we need to look at before moving on. You don't have to specify a parameter list if you don't want to use any parameters - but you still need the empty brackets! You don't have to assign the function a type in which case it defaults to int. A function doesn't have to return a value and the program that makes use of a function doesn't have to save any value it does return. For example, it is perfectly OK to use:

```
sum(1,2);
```

which simply throws away the result of adding 1 to 2. As this sort of thing offends some programmers you can use the data type void to indicate that a function doesn't return a value. For example:

```
void demo();
```

is a function with no parameters and no return value.

void is an ANSI C standard data type.

The break statement covered in a previous section can be used to exit a function. The break statement is usually linked with an if statement checking for a particular value. For example:

```
if (x==1) break;
```

If x contained 1 then the function would exit and return to the calling program.

Functions and Prototypes:

Where should a function's definition go in relation to the entire program - before or after main()? The only requirement is that the function's type has to be known before it is actually used. One way is to place the function definition earlier in the program than it is used - for example, before main(). The only problem is that most C programmers would rather put the main program at the top of the program listing. The solution is to declare the function separately at the start of the program. For example:

```
int sum();
main()
{
    etc...
```

declares the name sum to be a function that returns an integer. As long as you declare functions before they are used you can put the actual definition anywhere you like.

By default if you don't declare a function before you use it then it is assumed to be an int function - which is usually, but not always, correct. It is worth getting into the habit of putting function declarations at the start of your programs because this makes them easier to convert to full ANSI C.

What is ANSI C?:

When C was first written the standard was set by its authors Kernighan and Ritchie - hence "K&R C". In 1990, an international ANSI standard for C was established which differs from K&R C in a number of ways.

The only really important difference is the use of function prototypes. To allow the compiler to check that you are using functions correctly ANSI C allows you to include a function prototype which gives the type of the function and the type of each parameter before you define the function. For example, a prototype for the sum function would be:

```
int sum(int,int);
```

meaning sum is an int function which takes two int parameters. Obviously, if you are in the habit of declaring functions then this is a small modification. The only other major change is that you can declare parameter types along with the function as in:

```
int sum(int a, int b);  
{
```

rather than:

```
int sum(a,b)  
int a,b;  
{
```

was used in the original K&R C. Again, you can see that this is just a small change. Notice that even if you are using an ANSI compiler you don't have to use prototypes and the K&R version of the code will work perfectly well.

The Standard Library Functions:

Some of the "commands" in C are not really "commands" at all but are functions. For example, we have been using printf and scanf to do input and output, and we have used rand to generate random numbers - all three are functions.

There are a great many standard functions that are included with C compilers and while these are not really part of the language, in the sense that you can re-write them if you really want to, most C programmers think of them as fixtures and fittings. Later in the course we will look into the mysteries of how C gains access to these standard functions and how we can extend the range of the standard library. But for now a list of the most common libraries and a brief description of the most useful functions they contain follows:

stdio.h: I/O functions:

- getchar() returns the next character typed on the keyboard.
- putchar() outputs a single character to the screen.
- printf() as previously described
- scanf() as previously described

string.h: String functions

- strcat() concatenates a copy of str2 to str1
- strcmp() compares two strings
- strcpy() copies contents of str2 to str1

ctype.h: Character functions

- isdigit() returns non-0 if arg is digit 0 to 9
- isalpha() returns non-0 if arg is a letter of the alphabet
- isalnum() returns non-0 if arg is a letter or digit
- islower() returns non-0 if arg is lowercase letter
- isupper() returns non-0 if arg is uppercase letter

math.h: Mathematics functions

- acos() returns arc cosine of arg
- asin() returns arc sine of arg
- atan() returns arc tangent of arg
- cos() returns cosine of arg

exp() returns natural logarithm e
fabs() returns absolute value of num
sqrt() returns square root of num

time.h: Time and Date functions

time() returns current calendar time of system
difftime() returns difference in secs between two times
clock() returns number of system clock cycles since program execution

stdlib.h: Miscellaneous functions

malloc() provides dynamic memory allocation, covered in future sections
rand() as already described previously
srand() used to set the starting point for rand()

Throwing The Dice:

As an example of how to use functions, we conclude this section with a program that, while it isn't state of the art, does show that there are things you can already do with C. It also has to be said that some parts of the program can be written more neatly with just a little more C - but that's for later. All the program does is to generate a random number in the range 1 to 6 and displays a dice face with the appropriate pattern.

The main program isn't difficult to write because we are going to adopt the traditional programmer's trick of assuming that any function needed already exists. This approach is called stepwise refinement, and although its value as a programming method isn't clear cut, it still isn't a bad way of organising things:

```
main()
{
    int r;
    char ans;

    ans = getans();

    while(ans == 'y')
    {
        r = randn(6);
        blines(25);
        if (r==1) showone();
        if (r==2) showtwo();
        if (r==3) showthree();
        if (r==4) showfour();
        if (r==5) showfive();
        if (r==6) showsix();
        blines(21);
        ans = getans();
    }

    blines(2);
}
```

If you look at main() you might be a bit mystified at first. It is clear that the list of if statements pick out one of the functions showone, showtwo etc. and so these must do the actual printing of the dot patterns - but what is blines, what is getans and why are we using randn()? The last time we used a random number generator it was called rand()!

The simple answers are that blines(n) will print n blank lines, getans() asks the user a question and waits for the single letter answer, and randn(n) is a new random number generator function that produces a random integer in the range 1 to n - but to know this you would have written the main program. We decided what functions would make our task easier and named them. The next step is to write the code to fill in the details of each of the functions. There is nothing to stop me assuming that other functions that would make my job easier already exist. This is the main principle of stepwise refinement - never write any code if you can possibly invent another function! Let's start with randn().

This is obviously an int function and it can make use of the existing rand() function in the standard library

```
int randn(int n)
{
    return rand()%n + 1;
}
```

The single line of the body of the function just returns the remainder of the random number after dividing by n - % is the remainder operator - plus 1. An alternative would be to use a temporary variable to store the result and then return this value. You can also use functions within the body of other functions.

Next getans()

```
char getans()
{
    int ans;

    printf("Throw y/n ?");
    ans = -1;
    while (ans == -1)
    {
        ans=getchar();
    }
    return ans;
}
```

This uses the standard int function getchar() which reads the next character from the keyboard and returns its ASCII code or -1 if there isn't a key pressed. This function tends to vary in its behaviour according to the implementation you are using. Often it needs a carriage return pressed before it will return anything - so if you are using a different compiler and the program just hangs, try pressing "y" followed the by Enter or Return key.

The blines(n) function simply has to use a for loop to print the specified number of lines:

```
void blines(int n)
{
    int i;

    for(i=1 ; i<=n ; i++) printf("\n");
}
```

Last but not least are the functions to print the dot patterns. These are just boring uses of printf to show different patterns. Each function prints exactly three lines of dots and uses blank lines if necessary. The reason for this is that printing 25 blank lines should clear a standard text screen and after printing three lines printing 21 blank lines will scroll the pattern to the top of the screen. If this doesn't happen on your machine make sure you are using a 29 line text mode display.

```
void showone()
{
    printf("\n * \n");
}
```

```
void showtwo()
{
    printf(" * \n\n");
    printf(" * \n");
}
```

```
void showthree()
{
    printf(" * * \n");
    printf(" * * \n");
}
```

```

    printf(" * \n");
}

void showfour()
{
    printf(" * * \n\n");
    printf(" * * \n");
}

void showfive()
{
    printf(" * * \n");
    printf(" * \n");
    printf(" * * \n");
}

void showsix()
{
    int i;

    for(i=1 ; i>=3 ; i++) printf(" * * \n");
}

```

The only excitement in all of this is the use of a for loop in showsix! Type this all in and add:

```

void showone();
void showtwo();
void showthree();
void showfour();
void showfive();
void showsix();
int randn();
char getans();
void blines();

```

before the main function if you type the other functions in after.

Once you have the program working try modifying it. For example, see if you can improve the look of the patterns. You might also see if you can reduce the number of showx functions in use - the key is that the patterns are built up of combinations of two horizontal dots and one centered dot. Best of luck.

Data Types Part II

Objectives:

So far we have looked at local variable now we switch our attention to other types of variables supported by the C programming language:

- 1.Global Variables
- 2.Constant Data Types

Global variables:

Variables can be declared as either local variables which can be used inside the function it has been declared in (more on this in further sections) and global variables which are known throughout the entire program. Global variables are created by declaring them outside any function. For example:

```
int max;

main()
{
    .....
}
f1()
{
    .....
}
```

The int max can be used in both main and function f1 and any changes made to it will remain consistent for both functions. The understanding of this will become clearer when you have studied the section on functions but I felt I couldn't complete a section on data types without mentioning global and local variables.

Constant Data Types:

Constants refer to fixed values that may not be altered by the program. All the data types we have previously covered can be defined as constant data types if we so wish to do so. The constant data types must be defined before the main function. The format is as follows:

```
#define CONSTANTNAME value
```

for example:

```
#define SALESTAX 0.05
```

The constant name is normally written in capitals and does not have a semi-colon at the end. The use of constants is mainly for making your programs easier to be understood and modified by others and yourself in the future. An example program now follows:

```
#define SALESTAX 0.05
#include <stdio.h>
main()
{
    float amount, taxes, total;
    printf("Enter the amount purchased : ");
    scanf("%f",&amount);
    taxes = SALESTAX*amount;
    printf("The sales tax is £%4.2f",taxes);
    printf("\n The total bill is £%5.2f",total);
}
```

The float constant SALESTAX is defined with value 0.05. Three float variables are declared amount, taxes and total.

Display message to the screen is achieved using printf and user input handled by scanf. Calculation is then performed and results sent to the screen. If the value of SALESTAX alters in the future it is very easy to change the value where it is defined rather than go through the whole program changing the individual values separately, which would be very time consuming in a large program with several references. The program is also improved when using constants rather than values as it improves the clarity.

Arrays

Objectives:

Having read this section you should have a good understanding of the use of arrays in C.

Advanced Data Types:

Programming in any language takes a quite significant leap forwards as soon as you learn about more advanced data types - arrays and strings of characters. In C there is also a third more general and even more powerful advanced data type - the pointer but more about that later. In this section we introduce the array, but the first question is, why bother?

There are times when we need to store a complete list of numbers or other data items. You could do this by creating as many individual variables as would be needed for the job, but this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```
main()
{
  int a1,a2,a3,a4,a5;
  scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
  printf("%d %d %d %d %d",a5,a4,a3,a2,a1);
}
```

Doesn't look very pretty does it, and what if the problem was to read in 100 or more values and print them in reverse order? Of course the clue to the solution is the use of the regular variable names a1, a2 and so on. What we would really like to do is to use a name like a[i] where i is a variable which specifies which particular value we are working with. This is the basic idea of an array and nearly all programming languages provide this sort of facility - only the details alter.

In the case of C you have to declare an array before you use it - in the same way you have to declare any sort of variable. For example,

```
int a[5];
```

declares an array called a with five elements. Just to confuse matters a little the first element is a[0] and the last a[4]. C programmer's always start counting at zero! Languages vary according to where they start numbering arrays. Less technical, i.e. simpler, languages start counting from 1 and more technical ones usually start counting from 0. Anyway, in the case of C you have to remember that

```
type array[size]
```

declares an array of the specified type and with size elements. The first array element is array[0] and the last is array[size-1].

Using an array, the problem of reading in and printing out a set of values in reverse order becomes simple:

```
main()
{
  int a[5];
  int i;
  for(i=0;i < 5; ++i) scanf("%d",&a[i]);
  for(i=4;i > =0;--i) printf("%d",a[i]);
}
```

Well we said simple but I have to admit that the pair of for loops looks a bit intimidating. The for loop and the array data type were more or less made for each other. The for loop can be used to generate a sequence of values to pick out and process each element in an array in turn. Once you start using arrays, for loops like:

```
for (i=0 ; i<5 ; ++i)
```

to generate values in the order 0,1,2 and so forth, and

```
for(i=4;i>=0;--i)
```

to generate values in the order 4,3,2... become very familiar.

In Dis-array:

An array of character variables is in no way different from an array of numeric variables, but programmers often like to think about them in a different way. For example, if you want to read in and reverse five characters you could use:

```
main()
{
    char a[5];
    int i;
    for(i=0; i<5; ++i) scanf("%c",&a[i]);
    for(i=4;i>=0;--i) printf("%c",a[i]);
}
```

Notice that the only difference, is the declared type of the array and the %c used to specify that the data is to be interpreted as a character in scanf and printf. The trouble with character arrays is that to use them as if they were text strings you have to remember how many characters they hold. In other words, if you declare a character array 40 elements long and store H E L L O in it you need to remember that after element 4 the array is empty. This is such a nuisance that C uses the simple convention that the end of a string of characters is marked by a null character. A null character is, as you might expect, the character with ASCII code 0. If you want to store the null character in a character variable you can use the notation \0 - but most of the time you don't have to actually use the null character. The reason is that C will automatically add a null character and store each character in a separate element when you use a string constant. A string constant is indicated by double quotes as opposed to a character constant which is indicated by a single quote. For example:

```
"A"
```

is a string constant, but

```
'A'
```

is a character constant. The difference between these two superficially similar types of text is confusing at first and the source of many errors. All you have to remember is that "A" consists of two characters, the letter A followed by \0 whereas 'A' is just the single character A. If you are familiar with other languages you might think that you could assign string constants to character arrays and work as if a string was a built-in data type. In C however the fundamental data type is the array and strings are very much grafted on. For example, if you try something like:

```
char name[40];
name="Hello"
```

it will not work. However, you can print strings using printf and read them into character arrays using scanf. For example,

```
main()
{
    static char name[40] ="hello";

    printf("%s",name);
    scanf("%s",name);
    printf("%s",name);
}
```



```
}
```

This program reads in the text that you type, terminating it with a null and stores it in the character array name. It then prints the character array treating it as a string, i.e. stopping when it hits the first null string. Notice the use of the "%s" format descriptor in scanf and printf to specify that what is being printed is a string.

At this point the way that strings work and how they can be made a bit more useful and natural depends on understanding pointers which is covered in the next section.

Pointers

Objectives:

Having read this section you should be able to:

1. program using pointers
2. understand how C uses pointers with arrays

Point to Point:

Pointers are a very powerful, but primitive facility contained in the C language. Pointers are a throwback to the days of low-level assembly language programming and as a result they are sometimes difficult to understand and subject to subtle and difficult-to-find errors. Still it has to be admitted that pointers are one of the great attractions of the C language and there will be many an experienced C programmer spluttering and fuming at the idea that we would dare to refer to pointers as 'primitive'!

In an ideal world we would avoid telling you about pointers until the very last minute, but without them many of the simpler aspects of C just don't make any sense at all. So, with apologies, let's get on with pointers.

A variable is an area of memory that has been given a name. For example:

```
int x;
```

is an area of memory that has been given the name x. The advantage of this scheme is that you can use the name to specify where to store data. For example:

```
x=10;
```

is an instruction to store the data value 10 in the area of memory named x. The variable is such a fundamental idea that using it quickly becomes second nature, but there is another way of working with memory.

The computer access its own memory not by using variable names but by using a memory map with each location of memory uniquely defined by a number, called the address of that memory location.

A pointer is a variable that stores this location of memory. In more fundamental terms, a pointer stores the address of a variable . In more picturesque terms, a pointer points to a variable.

A pointer has to be declared just like any other variable - remember a pointer is just a variable that stores an address. For example,

```
int *p;
```

is a pointer to an integer. Adding an asterisk in front of a variable's name declares it to be a pointer to the declared type. Notice that the asterisk applies only to the single variable name that it is in front of, so:

```
int *p , q;
```

declares a pointer to an int and an int variable, not two pointers.

Once you have declared a pointer variable you can begin using it like any other variable, but in practice you also need to know the meaning of two new operators: & and *. The & operator returns the address of a variable. You can remember this easily because & is the 'A'mpersand character and it gets you the 'A'ddress. For example:

```
int *p , q;
```

declares p, a pointer to int, and q an int and the instruction:

```
p=&q;
```

stores the address of q in p. After this instruction you can think of p as pointing at q. Compare this to:

```
p=q;
```

which attempts to store the value in q in the pointer p - something which has to be considered an error.

The second operator * is a little more difficult to understand. If you place * in front of a pointer variable then the result is the value stored in the variable pointed at. That is, p stores the address, or pointer, to another variable and *p is the value stored in the variable that p points at.

The * operator is called the dereferencing operator and it helps not to confuse it with multiplication or with its use in declaring a pointer.

This multiple use of an operator is called operator overload.

Confused? Well most C programmers are confused when they first meet pointers. There seems to be just too much to take in on first acquaintance. However there are only three basic ideas:

- 1.To declare a pointer add an * in front of its name.
- 2.To obtain the address of a variable use & in front of its name.
- 3.To obtain the value of a variable use * in front of a pointer's name.

Now see if you can work out what the following means:

```
int *a , b , c;  
b = 10;  
a = &b;  
c = *a;
```

Firstly three variables are declared - a (a pointer to int), and b and c (both standard integers). The instruction stores the value 10 in the variable b in the usual way. The first 'difficult' instruction is a=&b which stores the address of b in a. After this a points to b.

Finally c = *a stores the value in the variable pointed to by a in c. As a points to b, its value i.e. 10 is stored in c. In other words, this is a long winded way of writing

```
c = b;
```

Notice that if a is an int and p is a pointer to an int then

```
a = p;
```

is nonsense because it tries to store the address of an int, i.e. a pointer value, in an int. Similarly:

```
a = &p;
```

tries to store the address of a pointer variable in a and is equally wrong! The only assignment between an int and a pointer to int that makes sense is:

```
a = *p;
```

Swap Shop:

At the moment it looks as if pointers are just a complicated way of doing something we can already do by a simpler method. However, consider the following simple problem - write a function which swaps the contents of two variables. That is, write swap(a,b) which will swap over the contents of a and b. In principle this should be easy:

```
function swap(int a , int b);  
{  
  int temp;  
  temp = a;  
  a    = b;  
  b    = temp;  
}
```

the only complication being the need to use a third variable temp to hold the value of a while the value of b overwrites it. However, if you try this function you will find that it doesn't work. You can use it - swap(a,b); - until you are blue in the face, but it just will not change the values stored in a and b back in the calling program. The reason is that all parameters in C are passed by value. That is, when you use swap(a,b) function the values in a and b are passed into the function swap via the parameters and any changes that are made to the parameters do not alter a and b back in the main program. The function swap does swap over the values in a and b within the function, but doesn't do so in the main program.

The solution to this very common problem is to pass not the values stored in the variables, but the addresses of the variables. The function can then use pointers to get at the values in the variables in the main program and modify them. That is, the function should be:

```
function swap(int *a , int *b);
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Notice that now the two parameters a and b are pointers and the assignments that effect the swap have to use the dereference operator to make sure that it is the values of the variables pointed at that are swapped. You should have no difficulty with:

```
temp = *a;
```

this just stores the value pointed at by a into temp. However,

```
*a = *b;
```

is a little more unusual in that it stores that value pointed at by b in place of the value pointed at by a. There is one final complication. When you use swap you have to remember to pass the addresses of the variables that you want to swap. That is not:

```
swap(a,b)
```

but

```
swap(&a,&b)
```

The rule is that whenever you want to pass a variable so that the function can modify its contents you have to pass it as an address. Equally the function has to be ready to accept an address and work with it. You can't take any old function and suddenly decide to pass it the address of a variable instead of its value. If you pass an address to a function that isn't expecting it the result is usually disaster and the same is true if you fail to pass an address to a function that is expecting one.

For example, calling swap as swap(a,b) instead of swap(&a,&b) will result in two arbitrary areas of memory being swapped over, usually with the result that the entire system, not just your program, crashes.

The need to pass an address to a function also explains the difference between the two I/O functions that we have been using since the beginning of this course. printf doesn't change the values of its parameters so it is called as printf("%d",a) but scanf does, because it is an input function, and so it is called as scanf("%d",&a).

Pointers And Arrays:

In C there is a very close connection between pointers and arrays. In fact they are more or less one and the same thing! When you declare an array as:

```
int a[10];
```

you are in fact declaring a pointer `a` to the first element in the array. That is, `a` is exactly the same as `&a[0]`. The only difference between `a` and a pointer variable is that the array name is a constant pointer - you cannot change the location it points at. When you write an expression such as `a[i]` this is converted into a pointer expression that gives the value of the appropriate element. To be more precise, `a[i]` is exactly equivalent to `*(a+i)` i.e. the value pointed at by `a + i`. In the same way `*(a + 1)` is the same as `a[1]` and so on.

Being able to add one to a pointer to get the next element of an array is a nice idea, but it does raise the question of what it means to add 'one' to a pointer. For example, in most implementations an `int` takes two memory locations and a `float` takes four. So if you declare an `int` array and add one to a pointer to it, then in fact the pointer will move on by two memory locations. However, if you declare a `float` array and add one to a pointer to it then the pointer has to move on by four memory locations. In other words, adding one to a pointer moves it on by an amount of storage depending on the type it is a pointer to.

This is, of course, precisely why you have to declare the type that the pointer is to point at! Only by knowing that `a` is a pointer to `int` and `b` is a pointer to `float` can the compiler figure out that

```
a + 1
```

means move the pointer on by two memory locations i.e. add 2, and

```
b + 1
```

means move the pointer on by four memory locations i.e. add 4. In practice you don't have to worry about how much storage a pointer's base type takes up. All you do need to remember is that pointer arithmetic works in units of the data type that the pointer points at. Notice that you can even use `++` and `--` with a pointer, but not with an array name because this is a constant pointer and cannot be changed. So to summarise:

1. An array's name is a constant pointer to the first element in the array that is `a==&a[0]` and `*a==a[0]`.
2. Array indexing is equivalent to pointer arithmetic - that is `a+i=&a[i]` and `*(a+i)==a[i]`.

It is up to you whether you want to think about an array as an array or an area of storage associated with a constant pointer. The view of it as an array is the more sophisticated and the further away from the underlying way that the machine works. The view as a pointer and pointer arithmetic is more primitive and closer to the hardware. In most cases the distinction is irrelevant and purely a matter of taste.

One final point connected with both arrays and functions is that when you pass an entire array to a function then by default you pass a pointer. This allows you to write functions that process entire arrays without having to pass every single value stored in the array - just a pointer to the first element. However, it also tempts you to write some very strange code unless you keep a clear head. Try the following - write a function that will fill an array with random values `randdat(a,n)` where `a` is the array and `n` is its size. Your first attempt might be something like:

```
void randdat(int *pa , int n)
{
    for (pa = 0 ; pa < n ; pa++ ) *pa = rand()%n + 1;
}
```

Well I hope your first attempt wouldn't be like this because it is wrong on a number of counts! The problem is that the idea of a pointer and the idea of an index have been confused. The pointer `pa` is supposed to point to the first element of the array, but the `for` loop sets it to zero and then increments it through a series of memory locations nowhere near the array. A lesser error is to suppose that `n-1` is the correct final value of the array pointer! As before, you will be lucky if this program doesn't crash the system, let alone itself! The correct way of doing the job is to use a `for` loop to step from 0 to `n-1`, but to use pointer arithmetic to access the correct array element:

```
int randdat(int *pa , int n)
{
    int i;
    for ( i=0 ; i< n ; ++i)
    {
```

```
    *pa = rand()%n + 1;
    ++pa;
}
```

Notice the way that the for loop looks just like the standard way of stepping through an array. If you want to make it look even more like indexing an array using a for loop you could write:

```
for(i=0 ; i<n ; ++i) *(pa+i)=rand()%n+1;
```

or even:

```
for(i=0 ; i<n ; ++i) pa[i]=rand()%n+1;
```

In other words, as long as you define pa as a pointer you can use array indexing notation with it and it looks as if you have actually passed an array. You can even declare a pointer variable using the notation:

```
int pa[];
```

that is, as an array with no size information. In this way the illusion of passing an array to a function is complete.

Objectives

This section brings together the use of two of C's fundamental data types, pointers and arrays, in the use of handling strings.

Having read this section you should be able to:

1. handle any string constant by storing it in an array.

Stringing Along:

Now that we have mastered pointers and the relationship between arrays and pointers we can take a second look at strings. A string is just a character array with the convention that the end of the valid data is marked by a null '\0'. Now you should be able to see why you can read in a character string using `scanf("%s", name)` rather than `scanf("%s",&name)` - name is already a pointer variable. Manipulating strings is very much a matter of pointers and special string functions. For example, the `strlen(str)` function returns the number of characters in the string str. It does this simply by counting the number of characters up to the first null in the character array - so it is important that you are using a valid null-terminated string. Indeed this is important with all of the C string functions.

You might not think that you need a function to copy strings, but simple assignment between string variables doesn't work. For example:

```
char a[10],b[10];
b = a;
```

does not appear to make a copy of the characters in a, but this is an illusion. What actually happens is that the pointer b is set to point to the same set of characters that a points to, i.e. a second copy of the string isn't created.

To do this you need `strcpy(a,b)` which really does make a copy of every character in a in the array b up to the first null character. In a similar fashion `strcat(a,b)` adds the characters in b to the end of the string stored in a. Finally there is the all-important `strcmp(a,b)` which compares the two strings character by character and returns true - that is 0 - if the results are equal.

Again notice that you can't compare strings using `a==b` because this just tests to see if the two pointers a and b are pointing to the same memory location. Of course if they are then the two strings are the same, but it is still possible for two strings to be the same even if they are stored at different locations.

You can see that you need to understand pointers to avoid making simple mistakes using strings. One last problem is how to initialise a character array to a string. You can't use:

```
a = "hello";
```

because a is a pointer and "hello" is a string constant. However, you can use:

```
strcpy(a,"hello")
```

because a string constant is passed in exactly the same way as a string variable, i.e. as a pointer. If you are worried where the string constant is stored, the answer is in a special area of memory along with all of the constants that the program uses. The main disadvantage of this method is that many compilers use an optimisation trick that results in only a single version of identical constants being stored. For example:

```
strcpy(b,"hello");
```

usually ends up with b pointing to the same string as a. In other words, this method isn't particularly safe!

A much better method is to use array initialisation. You can specify constants to be used to initialise any variable when it is declared. For example:

```
int a=10;
```

declares a to be an integer and initialises it to 10. You can initialise an array using a similar notation. For example:

```
int a[5] = {1,2,3,4,5};
```

declares an integer array and initialises it so that a[0]= 1, a[1] = 2 and so on. A character array can be initialised in the same way. For example:

```
char a[5]='h','e','l','l','o';
```

but a much better way is to write:

```
char a[6]="hello";
```

which also automatically stores a null character at the end of the string - hence a[6] and not a[5]. If you really want to be lazy you can use:

```
char a[] = "hello";
```

and let the compiler work out how many array elements are needed. Some compilers cannot cope with the idea of initialising a variable that doesn't exist for the entire life of the program. For those compilers to make initialisation work you need to add the keyword static to the front of the string declaration, therefore:

```
static char a[] = "hello";
```

As easy as... B or C?:

A few words of warning. If you are familiar with BASIC then you will have to treat C strings, and even C arrays, with some caution. They are not as easy or as obvious to use and writing a program that manipulates text is harder in C than in BASIC. If you try to use C strings as if it were BASIC strings you are sure to create some very weird and wonderful bugs!

A Sort Of Bubble Program:

This sections program implements a simple bubble sort - which is notorious for being one of the worst sorting methods known to programmer-kind, but it does have the advantage of being easy and instructive. Some of the routines have already been described in the main text and a range of different methods of passing data in functions have also been used.

The main routine is sort which repeats the scan function on the array until the variable done is set to 0. The scan function simply scans down the array comparing elements that are next door to each other. If they are in the wrong order then function swap is called to swap them over.

Study this program carefully with particular attention to the way arrays, array elements and variables are passed. It is worth saying that in some cases there are better ways of achieving the same results. In particular, it would have been easier not to use the variable done, but to have returned the state as the result of the scan function.

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

```
void randdat(int a[] , int n);  
void sort(int a[] , int n);  
void scan(int a[] , int n , int *done);  
void swap(int *a ,int *b);
```

```
main()  
{  
    int i;  
    int a[20];  
  
    randdat(a , 20);  
    sort(a , 20);
```



```
    for(i=0;i<20;++i) printf("%d\n" ,a[i]);  
}
```

```
void randdat(int a[1] , int n)  
{  
    int i;  
    for (i=0 ; i<n ; ++i)  
        a[i] = rand()%n+1;  
}
```

```
void sort(int a[1] , int n)  
{  
    int done;  
    done = 1;  
    while(done == 1) scan(a , n , &done);  
}
```

```
void scan(int a[1] , int n , int *done)  
{  
    int i;  
    *done=0;  
    for(i=0 ; i<n-1 ; ++i)  
    {  
        if(a[i]<a[i+1])  
        {  
            swap(&a[i],&a[i+1]);  
            *done=1;  
        }  
    }  
}
```

```
void swap(int *a ,int *b)  
{  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Structures

Objectives:

This section contains some very advanced but important features of the C programming language.

Having read this section you should be able to:

1. program using a structure rather than several arrays.
2. how pointer can be used in combination with structures to form linked list.

Structures:

The array is an example of a data structure. It takes simple data types like int, char or double and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of the same type. At first this seems perfectly reasonable. After all why would you want an array to be composed of twenty chars and two ints? Well this sort of mixture of data types working together is one of the most familiar of data structures. Consider for a moment a record card which records name, age and salary. The name would have to be stored as a string, i.e. an array of chars terminated with an ASCII null character, and the age and salary could be ints.

At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides struct. At first it is easier to think of this as a record - although it's a little more versatile than this suggests.

Defining A New Type:

Declaring a struct is a two-stage process. The first stage defines a new data type that has the required structure which can then be used to declare as many variables with the same structure as required. This two-stage process is often confusing at first - especially as it results in the need to think up multiple names with the same general meaning - but it really is quite simple. For example, suppose we need to store a name, age and salary as a single structure. You would first define the new data type using:

```
struct emprec
{
    char name[25];
    int age;
    int pay;
};
```

and then you would declare a new variable:

```
struct emprec employee
```

Notice that the new variable is called employee and it is of type emprec which has been defined earlier. You see what we mean about duplicating names - emprec is the name of the general employee record structure and employee is a particular example of this general type. It might help to compare the situation with that of a general int type and a particular int variable such as count - emprec is a type like int and employee is a variable like count. You can see that in general you can define a structure using:

```
struct name
{
    list of component variables
};
```

and you can have as long a list of component variables as you need. Once defined you can declare as many examples of the new type as you like using:

struct name list of variables;

For example:

```
struct emprec employee, oldemploy, newemploy;
```

and so on. If you want to you can also declare a structure variable within the type definition by writing its name before the final semi-colon. For example:

```
struct emprec  
{  
    char name[25];  
    int age;  
    int pay;  
} employee;
```

defines the structure and declares a structure variable called employee. The only trouble with this form is that not many C programmers use it and many will even think that it is an error! So how do we use a struct?

When you first start working with arrays it seems obvious that you access the individual elements of the array using an index as in a[i] for the ith element of the array, but how to get at the individual components of a structure? The answer is that you have to use qualified names. You first give the name of the structure variable and then the name of the component separated by a dot. For example, given:

```
struct emprec employee
```

then:

```
employee.age
```

is an int and:

```
employee.name
```

is a char array. Once you have used a qualified name to get down to the level of a component then it behaves like a normal variable of the type. For example:

```
employee.age=32;
```

is a valid assignment to an int and:

```
employee.name[2] = 'X';
```

is a valid assignment to an element of the char array. Notice that the qualified name uses the structure variable name and not the structure type name. You can also define a structure that includes another structure as a component and of course that structure can contain another structure and so on. In this case you simply use the name of each structure in turn, separated by dots, until you reach a final component that isn't a structure. For example, if you declare a struct firm which includes a component employee which is an emprec then:

```
firm.employee.age
```

is an int. You may be feeling a little disappointed at the way in which structures are used. When you first meet arrays it is obvious how useful they are because the array index is an integer which can be used within a loop to process vast amounts of data in a few lines of code. When you first meet the struct it just doesn't have the same obvious advantages. Because you have to write out a full qualified name to get at each of the components of the struct you can't automate the processing in the same way. However this is reasonable enough when you remember that each component of a struct can be a different data type! The point is that the value of a struct is different to that of an array. A struct can be used to wrap up a group of variables which form a coherent entity.

For example, C has no facilities for manipulating complex numbers but this is easy enough to put right using a struct and a few functions. A complex number is composed of two parts - a real and imaginary part - which can be implemented as single or double precision values. This suggests defining a

new struct type:

```
struct comp
{
    float real;
    float imag;
};
```

After this you can declare new complex variables using something like:

```
struct comp a,b;
```

The new complex variables cannot be used as if they were simple variables - because they are not. Most versions, of the C language do allow you to assign structures so you could write:

```
a=b;
```

as shorthand for

```
a.real=b.real;
a.imag=b.imag;
```

Being able to assign structures is even more useful when they are bigger. However you can't expect C to sort out what you mean by $c = a + b$ - for this you have to write out the rule for addition as:

```
c.real=a.real+b.real;
c.imag=a.imag+b.imag;
```

Structures and Functions:

Of course a sensible alternative to writing out the addition each time is to define a function to do the same job - but this raises the question of passing structures as parameters. Fortunately this isn't a big problem. Most C compilers, will allow you to pass entire structures as parameters and return entire structures. As with all C parameters structures are passed by value and so if you want to allow a function to alter a parameter you have to remember to pass a pointer to a struct. Given that you can pass and return structs the function is fairly easy:

```
struct comp add(struct comp a , struct comp b)
{
    struct comp c;
    c.real=a.real+b.real;
    c.imag=a.imag+ b.imag;
    return c;
}
```

After you have defined the add function you can write a complex addition as:

```
x=add(y,z)
```

which isn't too far from the $x=y+z$ that you would really like to use. Finally notice that passing a struct by value might use up rather a lot of memory as a complete copy of the structure is made for the function.

Pointers to Structures:

You can define a pointer to a structure in the same way as any pointer to any type. For example:

```
struct emprec *ptr
```

defines a pointer to an emprec. You can use a pointer to a struct in more or less the same way as any pointer but the use of qualified names makes it look slightly different For example:

```
(*ptr).age
```

is the age component of the emprec structure that ptr points at - i.e. an int. You need the brackets because '.' has a higher priority than '*'. The use of a pointer to a struct is so common, and the pointer notation so ugly, that there is an equivalent and more elegant way of writing the same thing. You can use:

```
prt->age
```

to mean the same thing as (*ptr).age. The notation gives a clearer idea of what is going on - prt points (i.e. ->) to the structure and .age picks out which component of the structure we want. Interestingly until C++ became popular the -> notation was relatively rare and given that many C text books hardly mentioned it this confused many experienced C programmers!

There are many reasons for using a pointer to a struct but one is to make two way communication possible within functions. For example, an alternative way of writing the complex number addition function is:

```
void comp add(struct comp *a , struct comp *b , struct comp *c)
{
    c->real=a->real+b->real;
    c->imag=a->imag+b->imag;
}
```

In this case c is now a pointer to a comp struct and the function would be used as:

```
add(&x,&y,&z);
```

Notice that in this case the address of each of the structures is passed rather than a complete copy of the structure - hence the saving in space. Also notice that the function can now change the values of x, y and z if it wants to. It's up to you to decide if this is a good thing or not!

Malloc:

Now we come to a topic that is perhaps potentially the most confusing. So far we have allowed the C compiler to work out how to allocate storage. For example when you declare a variable:

```
int a;
```

the compiler sorts out how to set aside some memory to store the integer. More impressive is the way that

```
int a[50]
```

sets aside enough storage for 50 ints and sets the name a to point to the first element. Clever though this may be it is just static storage. That is the storage is allocated by the compiler before the program is run - but what can you do if you need or want to create new variables as your program is running? The answer is to use pointers and the malloc function. The statement:

```
ptr=malloc(size);
```

reserves size bytes of storage and sets the pointer ptr to point to the start of it. This sounds excessively primitive - who wants a few bytes of storage and a pointer to it? You can make malloc look a little more appealing with a few cosmetic changes. The first is that you can use the sizeof function to allocate storage in multiples of a given type. For example:

```
sizeof(int)
```

returns a number that specifies the number of bytes needed to store an int. Using sizeof you can allocate storage using malloc as:

```
ptr= malloc(sizeof(int)*N)
```

where N is the number of ints you want to create. The only problem is what does ptr point at? The compiler needs to know what the pointer points at so that it can do pointer arithmetic correctly. In other words, the compiler can only interpret ptr++ or ptr=ptr+1 as an instruction to move on to the next int if it knows that the ptr is a pointer to an int. This works as long as you define the ptr to be a pointer to the type of variable that you want to work with. Unfortunately this raises the question of how malloc knows what the type of the pointer variable is - unfortunately it doesn't.

To solve this problem you can use a TYPE cast. This C play on words is a mechanism to force a value to a specific type. All you have to do is write the TYPE specifier in brackets before the value. So:

```
ptr = (*int) malloc(sizeof(int)*N)
```

forces the value returned by malloc to be a pointer to int. Now you can see how a simple idea ends up looking complicated. OK, so now we can acquire some memory while the program is running, but how can we use it? There are some simple ways of using it and some very subtle mistakes that you can make in trying to use it! For example, suppose during a program you suddenly decide that you need an int array with 50 elements. You didn't know this before the program started, perhaps because the information has just been typed in by the user. The easiest solution is to use:

```
int *ptr;
```

and then later on:

```
ptr = (*int) malloc(sizeof(int)*N)
```

where N is the number of elements that you need. After this definition you can use ptr as if it was a conventional array. For example:

```
ptr[i]
```

is the ith element of the array. The trap waiting for you to make a mistake is when you need a few more elements of the array. You can't simply use malloc again to get the extra elements because the block of memory that the next malloc allocates isn't necessarily next to the last lot. In other words, it might not simply tag on to the end of the first array and any assumption that it does might end in the program simply overwriting areas of memory that it doesn't own.

Another fun error that you are not protected against is losing an area of memory. If you use malloc to reserve memory it is vital that you don't lose the pointer to it. If you do then that particular chunk of memory isn't available for your program to use until it is restarted.

Structures and Linked Lists:

You may be wondering why malloc has been introduced right after the structure. The answer is that the dynamic allocation of memory and the struct go together a bit like the array and the for loop. The best way to explain how this all fits together is via a simple example. You can use malloc to create as many variables as you want as the program runs, but how do you keep track of them? For every new variable you create you also need an extra pointer to keep track of it. The solution to this otherwise tricky problem is to define a struct which has a pointer as one of its components. For example:

```
struct list
{
    int data;
    struct list *ptr;
};
```

This defines a structure which contains a single int and - something that looks almost paradoxical - a pointer to the structure that is being defined. All you really need to know is that this is reasonable and it works. Now if you use malloc to create a new struct you also automatically get a new pointer to the struct. The final part of the solution is how to make use of the pointers. If you start off with a single 'starter' pointer to the struct you can create the first new struct using malloc as:

```
struct list *star;
start = (*struct list) malloc(sizeof(list))
```

After this start points to the first and only example of the struct. You can store data in the struct using statements like:

```
start->data=value;
```

The next step is to create a second example of the struct:

```
start = (*struct list) malloc(sizeof(list));
```

This does indeed give us a new struct but we have now lost the original because the pointer to it has been overwritten by the pointer to the new struct. To avoid losing the original the simplest solution is to use:

```
struct list *start,newitem;
newitem = (*struct list) malloc(sizeof(list));
start->prt=start;
start=newitem;
```

This stores the location of the new struct in newitem. Then it stores the pointer to the existing struct into the newitem's pointer and sets the start of the list to be the newitem. Finally the start of the list is set to point at the new struct. This procedure is repeated each time a new structure is created with the result that a linked list of structures is created. The pointer start always points to the first struct in the list and the prt component of this struct points to the next and so on. You should be able to see how to write a program that examines or prints the data in each of the structures. For example:

```
thisptr=start;
while (1==1)
{
    printf("%d",thisprt-> data);
    thisprt=thisprt->prt;
}
```

This first sets thisptr to the start of the list, prints the data in the first element and then gets the pointer to the next struct in the list and so on. How does the program know it has reached the end of the list? At the moment it just keeps going into the deep and uncharted regions of your machine's memory! To stop it we have to mark the end of the list using a null pointer. Usually a pointer value of 0 is special in that it never occurs in a pointer pointing at a valid area of memory. You can use 0 to initialise a pointer so that you know it isn't pointing at anything real. So all we have to do is set the last pointer in the list to 0 and then test for it That is:

```
thisptr=start;
while (thisptr!=0)
{
    printf("%d",thisprt->data);
    thisprt=thisprt-> prt;
}
```

To be completely correct you should TYPE cast 0 to be a pointer to the struct in question. That is:

```
while (thisptr!=(struct list*)0)
```

By generally mucking about with pointers stored in the list you can rearrange it, access it, sort it, delete items and do anything you want to. Notice that the structures in the list can be as complicated as you like and, subject to there being enough memory, you can create as many structures as you like.

You can use the same sort of technique to create even more complicated list structures. For example you can

introduce another pointer into each structure and a pointer to the end of the list so that you can work your way along it in the other direction - a doubly linked list. You can create stacks, queues, deques, trees and so on. The rest of the story is a matter of either inventing these data structures for yourself or looking them up in a suitable book.

Structures and C++:

The reason why structures are even more important for today's budding C programmer is that they turn into classes in C++. A class is a structure where you can define components that are functions. In this case the same distinction between a data TYPE and an example of the TYPE, i.e. a variable, is maintained only now the instances of the class include functions as well as data. The same qualified naming system applies to the class and the use of pointers and the -> operator. As this is the basis of C++'s object-oriented features it is important to understand.

Header Files:

The final mystery of C that needs to be discussed is the header file. This started off as a simple idea, a convenience to make programming easier. If you have a standard set of instructions that you want to insert in a lot of programs that you are writing then you can do it using the #include statement.

The # symbol at the start indicates that this isn't a C statement but one for the C pre-processor which looks at the text file before the compiler gets it. The #include tells the pre-processor to read in a text file and treat it as if it was part of the program's text. For example:

```
#include "copy.txt"
```

could be used to include a copyright notice stored in the file copy.txt. However the most common use of the #include is to define constants and macros. The C pre-processor is almost a language in its own right. For example, if you define the identifier NULL as:

```
#define NULL 0
```

then whenever you use NULL in your program the pre-processor substitutes 0. In most cases you want these definitions to be included in all your programs and so the obvious thing to do is to create a separate file that you can #include.

This idea of using standard include files has spiralled out of all proportions. Now such include files are called header files and they are distinguished by ending in the extension .h. A header file is generally used to define all of the functions, variables and constants contained in any function library that you might want to use. The header file stdio.h should be used if you want to use the two standard I/O functions printf and scanf. The standard libraries have been covered in a previous section.

This sort of use of header files is simple enough but over time more and more standard elements of the C environment have been moved into header files. The result is that header files become increasingly mysterious to the beginner. Perhaps they reach their ultimate in complexity as part of the Windows development environment. So many constants and macros are defined in the Windows header files that they amount to hundreds of lines! As another example of how you could use a header file consider the complex structure defined earlier. At the moment it looks messy to declare a new complex variable as:

```
struct comp a,b;
```

If you want to make the complex TYPE look like other data types all you need is a single #define

```
#define COMPLEX struct comp
```

After this you can write:

```
COMPLEX a,b;
```

and the pre-processor will automatically replace COMPLEX by struct comp for you when you compile the program. Put this #define and any others needed to make the complex number type work and you have the makings of a complex.h header file of your very own.

File Handling

Objectives:

So far we have entered information into our programs via the computer's keyboard. This is somewhat laborious if we have a lot of data to process. The solution is to combine all the input data into a file and let our C program read the information when it is required.

Having read this section you should be able to:

- 1.open a file for reading or writing
- 2.read/write the contents of a file
- 3.close the file

The Stream File:

Although C does not have any built-in method of performing file I/O, the C standard library contains a very rich set of I/O functions providing an efficient, powerful and flexible approach.

We will cover the ANSI file system but it must be mentioned that a second file system based upon the original UNIX system is also used but not covered on this course.

A very important concept in C is the stream. In C, the stream is a common, logical interface to the various devices that comprise the computer. In its most common form, a stream is a logical interface to a file. As C defines the term "file", it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all streams are the same. The stream provides a consistent interface and to the programmer one hardware device will look much like another.

A stream is linked to a file using an open operation. A stream is disassociated from a file using a close operation. The current location, also referred to as the current position, is the location in a file where the next file access will occur. There are two types of streams: text (used with ASCII characters some character translation takes place, may not be one-to-one correspondence between stream and what's in the file) and binary (used with any type of data, no character translation, one-to-one between stream and file).

To open a file and associate it with a stream, use `fopen()`. Its prototype is shown here:

```
FILE *fopen(char *fname,char *mode);
```

The `fopen()` function, like all the file-system functions, uses the header `stdio.h`. The name of the file to open is pointed to by `fname` (must be a valid name). The string pointed at for `mode` determines how the file may be accessed as shown:

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append a binary file for read/write

If the open operation is successful, `fopen()` returns a valid file pointer. The type `FILE` is defined in `stdio.h`. It is a structure that holds various kinds of information about the file, such as size. The file pointer will be used with all other functions that operate on the file and it must never be altered or the object it points to. If `fopen()` fails it returns a `NULL` pointer so this must always be checked for when opening a file. For example:

```
FILE *fp;
```

```
if ((fp = fopen("myfile", "r")) == NULL){
    printf("Error opening file\n");
    exit(1);
}
```

To close a file, use `fclose()`, whose prototype is

```
int fclose(FILE *fp);
```

The `fclose()` function closes the file associated with `fp`, which must be a valid file pointer previously obtained using `fopen()`, and disassociates the stream from the file. The `fclose()` function returns 0 if successful and EOF (end of file) if an error occurs.

Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it using these two functions.

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
```

The `getc()` function reads the next byte from the file and returns it as an integer and if error occurs returns EOF. The `getc()` function also returns EOF when the end of file is reached. Your routine can assign `fgetc()`'s return value to a char you don't have to assign it to an integer.

The `fputc()` function writes the bytes contained in `ch` to the file associated with `fp` as an unsigned char. Although `ch` is defined as an int, you may call it using simply a char. The `fputc()` function returns the character written if successful or EOF if an error occurs.

Text File Functions:

When working with text files, C provides four functions which make file operations easier. The first two are called `fputs()` and `fgets()`, which write or read a string from a file, respectively. Their prototypes are:

```
int fputs(char *str, FILE *fp);
char *fgets(char *str, int num, FILE *fp);
```

The `fputs()` function writes the string pointed to by `str` to the file associated with `fp`. It returns EOF if an error occurs and a non-negative value if successful. The null that terminates `str` is not written and it does not automatically append a carriage return/linefeed sequence.

The `fgetc()` function reads characters from the file associated with `fp` into a string pointed to by `str` until `num-1` characters have been read, a newline character is encountered, or the end of the file is reached. The string is null-terminated and the newline character is retained. The function returns `str` if successful and a null pointer if an error occurs.

The other two file handling functions to be covered are `fprintf()` and `fscanf()`. These functions operate exactly like `printf()` and `scanf()` except that they work with files. Their prototypes are:

```
int fprintf(FILE *fp, char *control-string, ...);
int fscanf(FILE *fp, char *control-string ...);
```

Instead of directing their I/O operations to the console, these functions operate on the file specified by `fp`. Otherwise their operations are the same as their console-based relatives. The advantages to `fprintf()` and `fscanf()` is that they make it very easy to write a wide variety of data to a file using a text format.

Binary File Functions:

The C file system includes two important functions: `fread()` and `fwrite()`. These functions can read and write any type of data, using any kind of representation. Their prototypes are:

```
size_t fread(void *buffer, size_t size, size_t num, FILE *fp);
size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
```

The `fread()` function reads from the file associated with `fp`, `num` number of objects, each object size bytes long, into

buffer pointed to by buffer. It returns the number of objects actually read. If this value is 0, no objects have been read, and either end of file has been encountered or an error has occurred. You can use feof() or ferror() to find out which. Their prototypes are:

```
int feof(FILE *fp);
int ferror(FILE *fp);
```

The feof() function returns non-0 if the file associated with fp has reached the end of file, otherwise it returns 0. This function works for both binary files and text files. The ferror() function returns non-0 if the file associated with fp has experienced an error, otherwise it returns 0.

The fwrite() function is the opposite of fread(). It writes to file associated with fp, num number of objects, each object size bytes long, from the buffer pointed to by buffer. It returns the number of objects written. This value will be less than num only if an output error as occurred.

The void pointer is a pointer that can point to any type of data without the use of a TYPE cast (known as a generic pointer). The type size_t is a variable that is able to hold a value equal to the size of the largest object supported by the compiler. As a simple example, this program write an integer value to a file called MYFILE using its internal, binary representation.

```
#include <stdio.h> /* header file */
#include <stdlib.h>
void main(void)
{
    FILE *fp; /* file pointer */
    int i;

    /* open file for output */
    if ((fp = fopen("myfile", "w"))==NULL){
        printf("Cannot open file \n");
        exit(1);
    }
    i=100;

    if (fwrite(&i, 2, 1, fp) !=1){
        printf("Write error occurred");
        exit(1);
    }
    fclose(fp);

    /* open file for input */
    if ((fp =fopen("myfile", "r"))==NULL){
        printf("Read error occurred");
        exit(1);
    }
    printf("i is %d",i);
    fclose(fp);
}
```

File System Functions:

You can erase a file using remove(). Its prototype is

```
int remove(char *file-name);
```

You can position a file's current location to the start of the file using rewind(). Its prototype is

```
void rewind(FILE *fp);
```

Hopefully I have given you enough information to at least get you started with files. Its really rather easy once you get started.

Command Line Parameters:

Many programs allow command-line arguments to be specified when they are run. A command-line argument is the information that follows the program's name on the command line of the operating system. Command-line arguments are used to pass information to the program. For example, when you use a text editor, you probably specify the name of the file you want to edit after the name of the word processing program. For example, if you use a word processor called WP, then this line causes the file TEST to be edited.

WP TEST

Here, TEST is a command-line argument. Your C programs may also utilize command-line arguments. These are passed to a C program through two arguments to the main() function. The parameters are called argc and argv. These parameters are optional and are not used when no command-line arguments are being used.

The argc parameter holds the number of arguments on the command-line and is an integer. It will always be at least 1 because the name of the program qualifies as the first argument. The argv parameter is an array of string pointers. The most common method for declaring argv is shown here.

```
char *argv[];
```

The empty brackets indicate that it is an array of undetermined length. All command-line arguments are passed to main() as strings. To access an individual string, index argv. For example, argv[0] points to the program's name and argv[1] points to the first argument. This program displays all the command-line arguments that it is called with.

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

```
void main(int argc, char *argv[])
{
    int i;

    for (i=1; i<argc; i++) printf("%s",argv[i]);
}
```

The ANSI C standard does not specify what constitutes a command-line argument, because operating systems vary considerably on this point. However, the most common convention is as follows:

Each command-line argument must be separated by a space or a tab character. Commas, semicolons, and the like are not considered separators. For example:

This is a test

is made up of four strings, but

this,that,and,another

is one string. If you need to pass a command-line argument that does, in fact contain spaces, you must place it between quotes, as shown in this example:

```
"this is a test"
```

A further example of the use of argc and argv now follows:

```
void main(int argc, char *argv[])
{
    if (argc !=2) {
        printf("Specify a password");
        exit(1);
    }
    if (!strcmp(argv[1], "password"))
        printf("Access Permitted");
    else
    {
        printf("Access denied");
        exit(1);
    }
    program code here .....
}
```

This program only allows access to its code if the correct password is entered as a command-line argument. There are many uses for command-line arguments and they can be a powerful tool.

My final example program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If the file contains at least one of these characters, it reports this fact. This program uses argv to access the file name and the character for which to search.

```
/*Search specified file for specified character. */

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

void main(int argc, char *argv[])
{
    FILE *fp; /* file pointer */
    char ch;

    /* see if correct number of command line arguments */
    if(argc !=3) {
        printf("Usage: find <filename> <ch>\n");
        exit(1);
    }

    /* open file for input */
    if ((fp = fopen(argv[1], "r"))==NULL) {
        printf("Cannot open file \n");
        exit(1);
    }

    /* look for character */
    while ((ch = getc(fp)) !=EOF) /* where getc() is a */
        if (ch== *argv[2]) { /*function to get one char*/
            printf("%c found",ch); /* from the file */
            break;
        }
    fclose(fp);
}
```

The names of argv and argc are arbitrary - you can use any names you like. However, argc and argv have traditionally been used since C's origin. It is a good idea to use these names so that anyone reading your program can quickly identify them as command-line parameters.

Recommended Books

It is always difficult to recommend books for any course and none more so than a C programming course - there are plenty of books to choose from. It is important to get a book that suits your reading style and also provides you with plenty of worked examples.

A suggested reading list is as follows:

Title:The C Programming Language ANSI C Version

Author:Kernighan & Ritchie

Publisher:Prentice Hall Software Series

ISBN:0-13-110362-8

Comment:Ideal reference book and goes well beyond the level of this course. It is expensive but the serious C programmer will kick themselves if they don't buy this book.

Title:ANSI C - Made Easy

Author:Herbert Schildt

Publisher:Osborne McGraw-Hill

ISBN:0-07-881500-2

Comment:

Title:Learning to Program in C

Author:N. Kantaris

Publisher: Babani

ISBN:0-85934-203-4

Comment:A good cheap beginners guide.

Title:Illustrating ANSI C

Author:Donald Alcock

Publisher:Cambridge University Press

ISBN:0-521-42483-6

Comment:A good book for the mathematically inclined.

Title:C - The Complete Reference

Author:Herbert Schildt

Publisher:Osborne McGraw-Hill

ISBN:0-07-881263-1

Comment:

Title:Numerical Recipes in C

Author:W.H.Press, et al

Comment:An advanced level book with, as the name implies, ready-made solutions to your programming problems.

Objectives

This section is only for reference! It contains the following information:

- 1.Names of all C's Standard Libraries
- 2.The functions they contain

Input and Output: <stdio.h>

```
FILE *fopen(const char *filename, const char *mode)
FILE *freopen(const char *filename, const char *mode, FILE *stream)
int fflush(FILE *stream)
int fclose(FILE *stream)
int remove(const char *filename)
int rename(const char *oldname, const char *newname)
FILE *tmpfile(void)
char *tmpnam(char s[L_tmpnam])
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
void setbuf(FILE *stream, char *buf)
int fprintf(FILE *stream, const char *format, ...)
int sprintf(char *s, const char *format, ...)
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
int fscanf(FILE *stream, const char *format, ...)
int scanf(const char *format, ...)
int sscanf(char *s, const char *format, ...)
int fgetc(FILE *stream)
char *fgets(char *s, int n, FILE *stream)
int fputc(int c, FILE *stream)
int fputs(const char *s, FILE *stream)
int getc(FILE *stream)
int getchar(void)
char *gets(char *s)
int putc(int c, FILE *stream)
int putchar(int c)
int ungetc(int c, FILE *stream)
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)
int fseek(FILE *stream, long offset, int origin)
long ftell(FILE *stream)
void rewind(FILE *stream)
int fgetpos(FILE *stream, fpos_t *ptr)
int fsetpos(FILE *stream, const fpos_t *ptr)
void clearerr(FILE *stream)
int feof(FILE *stream)
int ferror(FILE *stream)
void perror(const char *s)
```

Character Class Tests: <ctype.h>

```
isalnum(c)
isalpha(c)
iscntrl(c)
isdigit(c)
isgraph(c)
islower(c)
isprint(c)
ispunct(c)
isspace(c)
isupper(c)
isxdigit(c)
```

String Functions: <string.h>

```
char *strcpy(s , ct)
char *strncpy(s , ct , n)
char *strcat(s , ct)
char *strncat(s , ct , n)
int strcmp(cs , ct)
int strncmp(cs , ct ,n)
char *strchr(cs , c)
char *strrchr(cs , c)
size_t strspn(cs , ct)
size_t strcspn(cs , ct)
char *strstr(cs , ct)
size_t strlen(cs)
char *strerror(n)
char *strtok(s , ct)
```

Mathematical Functions: <math.h>

```
sin(x)
cos(x)
tan(x)
asin(x)
acos(x)
atan(x)
atan2(x)
sinh(x)
cosh(x)
tanh(x)
exp(x)
log(x)
log10(x)
pow(x,y)
sqrt(x)
ceil(x)
floor(x)
fabs(x)
ldexp(x)
frexp(x,double *ip)
modf(x,double *ip)
fmod(x,y)
```

Utility Functions: <stdlib.h>

```
double atof(const char *s)
int atoi(const char *s)
long atol(const char *s)
double strtod(const char *s, char **endp)
long strtol(const char *s, char **endp, int base)
unsigned long strtoul(const char *s, char **endp, int base)
int rand(void)
void srand(unsigned int seed)
void *calloc(size_t nobj, size_t size)
void *malloc(size_t size)
void *realloc(void *p, size_t size)
void free(void *p)
void abort(void)
void exit(int status)
int atexit(void (*fcn)(void))
int system(const char *s)
char *getenv(const char *name)
void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp)(const void *keyval, const void *datum))
void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *))
int abs(int n)
long labs(long n)
div_t div(int num, int denom)
ldiv_t ldiv(long num , long denom)
```


Diagnostics: <assert.h>

```
void assert(int expression)
```

Non-local Jumps: <setjmp.h>

```
int setjmp(jmp_buf env)
void longjmp(jmp_buf env, int val)
```

Signals: <signal.h>

```
void (*signal(int sig, void (*handler)(int)))(int)
```

Data and Time Functions: <time.h>

```
clock_t clock(void)
time_t time(time_t , *tp)
double difftime(time_t time2 , time_t time1)
time_t mktime(struct tm *tp)
char *asctime(const time_t *tp)
char *ctime(const time_t *tp)
struct tm *gmtime(const time_t *tp)
struct tm *localtime(const time_t *tp)
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
```

Editor Shortcuts

Cursor:

These commands control how the cursor behaves.

Command	Description
Left Arrow	Left one character
Right Arrow	Right one character
Up Arrow	Up one line
Down Arrow	Down one line
Control + Left	Left one word
Control + Right	Right one word
Home	Start of line
End	End of line
Page Up	Up one page
Page Down	Down one page
	Left one page
	Right one page
Control + Page Up	Top of page
Control + Page Down	Bottom of page
Control + Home	Abs begin
Control + End	Abs end
	Move specified coordinates

Selection:

These commands control how the currently highlighted text behaves.

Command	Description
Shift + Left	Select left
Shift + Right	Select right
Shift + Up	Select up
Shift + Down	Select down
Control + Shift + Left	Select word left
Control + Shift + Right	Select word right
Shift + Home	Select line start
Shift + End	Select line end
Shift + Page Up	Select page up
Shift + Page Down	Select page down
	Select page left
	Select page right
Control + Shift + Page Up	Select page top
Control + Shift + Page Down	Select page bottom
Control + Shift + Home	Select editor top
Control + Shift + End	Select gotoxy
Control + A	Select All
Control + Insert	Copy select to clip
Control + C	

Scrolling:

These commands control everything to do with scrolling.

Command	Description
Scroll Up + Control	Go up one line
Scroll Down + Control	Go down one line
Scroll Left	Left one character
Scroll Right	Right one character

Modes:

These commands control modes

Command	Description
Insert	Set insert mode
Control + Shift + N	Set overwrite mode
Control + Shift + C	Toggle insert/overwrite
Control + Shift + L	Selection type is normal
	Selection type is column
	Selection type is line

Actions:

These commands perform various actions.

Command	Description
Control + Shift + B	Go to matching bracket
Control + (number)	Move to marker (number)
Control + Shift + (number)	Set marker (number)
F1	Context sensitive help on word

Delete:

All the commands having to do with deleting shit.

Command	Description
Backspace	Character to left
Shift + Backspace	
Delete	Character to right
Control + T	Word to right
Control + Backspace	Word to left
	From cursor to start of line
Control + Shift + Y	From cursor to end of line
Control + Y	Current line
Enter	Everything in editor
Shift + Enter	Line break at current position, move caret
Control + M	
Control + N	Line break at current position, no move caret
	Insert character at current position
Alt + Backspace	Perform undo if available
Control + Z	Perform undo if available
Alt + Shift + Back	Perform redo if available
Control + Shift + Z	Perform redo if available
Shift + Delete	Remove selection place on clipboard
Control + X	Remove selection place on clipboard
Shift + Insert	Move clipboard contents to current position
Control + V	Move clipboard contents to current position
Control + Shift + I	Move selection to right
Control + Shift + U	Move selection to left
Tab	Tab key
Shift + Tab	Tab to left

Resources

Links:

{button Bloodshed.net} The home of Bloodshed Software...

{button Mingw.org} The home of the MinGW compiler...

{button Dev-C++ Source Code} Dev-C++ source code...

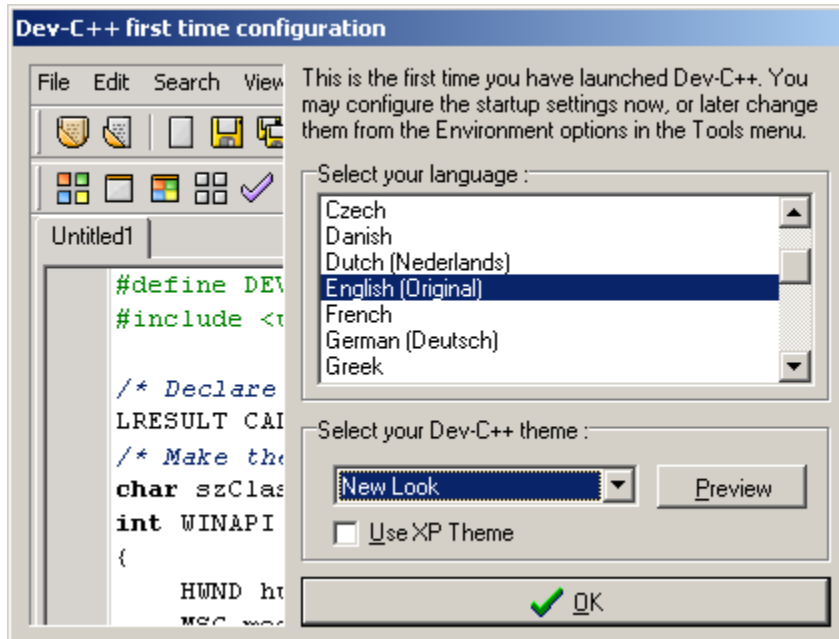
{button Mingw Source Code} Dev-C++ source code...

{button Dev-C++ Software Forum} Dev-C++ Forum...

{button Dev-C++ Developer Forum} Dev-C++ Developer Forum...

Getting Started

When you launch Dev-C++ for the first time, you will see the following dialog :



You can select the language Dev-C++ will use, as well as the icon theme. To add a XP flavor to Dev-C++, check the 'Use XP Theme' box.

If you installed a Dev-C++ version which include a Mingw compiler system, this is all you need to do to configure the software, since all compiler paths have been set to default. In this case you can then proceed to the [Basic Steps](#) `ID_CREATEPROJECT`.

However, if you prefer to use your own GCC distribution (like Cygwin), you can manually configure the compiler and directories settings to feet your own system. You can do that in [Compiler Options](#) `ID_COMPILEROPTIONS`.

ID_CREATEPROJECT

