An Introduction to GCC

for the GNU Compilers `gcc` and `g++`
Revised and updated

Brian Gough
Foreword by Richard M. Stallman

A catalogue record for this book is available from the British Library.

Second printing, August 2005 (1/8/2005). Revised and updated.
First printing, March 2004 (7/3/2004).

Published by Network Theory Limited.

15 Royal Park
Bristol
BS8 3AL
United Kingdom

Email: info@network-theory.co.uk

ISBN 0-9541617-9-3

Further information about this book is available from
http://www.network-theory.co.uk/gcc/intro/

Cover Image: From a layout of a fast, energy-efficient hardware stack.(1) Image created with the free Electric VLSI design system by Steven Rubin of Static Free Software (www.staticfreesoft.com). Static Free Software provides support for Electric to the electronics design industry.

Copyright © 2004, 2005 Network Theory Ltd.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being "A Network Theory Manual", and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License".

(a) The Back-Cover Text is: "The development of this manual was funded entirely by Network Theory Ltd. Copies published by Network Theory Ltd raise money for more free documentation."

---

(1) "A Fast and Energy-Efficient Stack" by J. Ebergen, D. Finchelstein, R. Kao, J. Lexau and R. Hopkins.

Table of Contents

Foreword

This foreword has been kindly contributed by Richard M. Stallman, the principal author of GCC and founder of the GNU Project.

This book is a guide to getting started with GCC, the GNU Compiler Collection. It will tell you how to use GCC as a programming tool. GCC is a programming tool, that's true—but it is also something more. It is part of a 20-year campaign for freedom for computer users.

We all want good software, but what does it mean for software to be "good"? Convenient features and reliability are what it means to be *technically* good, but that is not enough. Good software must also be *ethically* good: it has to respect the users' freedom.

As a user of software, you should have the right to run it as you see fit, the right to study the source code and then change it as you see fit, the right to redistribute copies of it to others, and the right to publish a modified version so that you can contribute to building the community. When a program respects your freedom in this way, we call it *free software*. Before GCC, there were other compilers for C, Fortran, Ada, etc. But they were not free software; you could not use them in freedom. I wrote GCC so we could use a compiler without giving up our freedom.

A compiler alone is not enough—to use a computer system, you need a whole operating system. In 1983, all operating systems for modern computers were non-free. To remedy this, in 1984 I began developing the GNU operating system, a Unix-like system that would be free software. Developing GCC was one part of developing GNU.

By the early 90s, the nearly-finished GNU operating system was completed by the addition of a kernel, Linux, that became free software in 1992. The combined GNU/Linux operating system has achieved the goal of making it possible to use a computer in freedom. But freedom is never automatically secure, and we need to work to defend it. The Free Software Movement needs your support.

Richard M. Stallman
February 2004

1 Introduction

The purpose of this book is to explain the use of the GNU C and C++ compilers, `gcc` and `g++`. After reading this book you should understand how to compile a program, and how to use basic compiler options for optimization and debugging. This book does not attempt to teach the C or C++ languages themselves, since this material can be found in many other places (see [Further reading], page 111).

Experienced programmers who are familiar with other systems, but new to the GNU compilers, can skip the early sections of the chapters "*Compiling a C program*", "*Using the preprocessor*" and "*Compiling a C++ program*". The remaining sections and chapters should provide a good overview of the features of GCC for those already know how to use other compilers.

1.1 A brief history of GCC

The original author of the GNU C Compiler (GCC) is Richard Stallman, the founder of the GNU Project.

The GNU Project was started in 1984 to create a complete Unix-like operating system as free software, in order to promote freedom and cooperation among computer users and programmers. Every Unix-like operating system needs a C compiler, and as there were no free compilers in existence at that time, the GNU Project had to develop one from scratch. The work was funded by donations from individuals and companies to the Free Software Foundation, a non-profit organization set up to support the work of the GNU Project.

The first release of GCC was made in 1987. This was a significant breakthrough, being the first portable ANSI C optimizing compiler released as free software. Since that time GCC has become one of the most important tools in the development of free software.

A major revision of the compiler came with the 2.0 series in 1992, which added the ability to compile C++. In 1997 an experimental branch of the compiler (EGCS) was created, to improve optimization and C++ support. Following this work, EGCS was adopted as the new main-line of GCC development, and these features became widely available in the 3.0 release of GCC in 2001.

Over time GCC has been extended to support many additional languages, including Fortran, ADA, Java and Objective-C. The acronym GCC is now used to refer to the "GNU Compiler Collection". Its development is guided by the *GCC Steering Committee*, a group composed of representatives from GCC user communities in industry, research and academia.

1.2 Major features of GCC

This section describes some of the most important features of GCC.

First of all, GCC is a portable compiler—it runs on most platforms available today, and can produce output for many types of processors. In addition to the processors used in personal computers, it also supports microcontrollers, DSPs and 64-bit CPUs.

GCC is not only a native compiler—it can also *cross-compile* any program, producing executable files for a different system from the one used by GCC itself. This allows software to be compiled for embedded systems which are not capable of running a compiler. GCC is written in C with a strong focus on portability, and can compile itself, so it can be adapted to new systems easily.

GCC has multiple language *frontends*, for parsing different languages. Programs in each language can be compiled, or cross-compiled, for any architecture. For example, an ADA program can be compiled for a microcontroller, or a C program for a supercomputer.

GCC has a modular design, allowing support for new languages and architectures to be added. Adding a new language front-end to GCC enables the use of that language on any architecture, provided that the necessary run-time facilities (such as libraries) are available. Similarly, adding support for a new architecture makes it available to all languages.

Finally, and most importantly, GCC is free software, distributed under the GNU General Public License (GNU GPL).[1] This means you have the freedom to use and to modify GCC, as with all GNU software. If you need support for a new type of CPU, a new language, or a new feature you can add it yourself, or hire someone to enhance GCC for you. You can hire someone to fix a bug if it is important for your work.

Furthermore, you have the freedom to share any enhancements you make to GCC. As a result of this freedom you can also make use of enhancements to GCC developed by others. The many features offered by GCC today show how this freedom to cooperate works to benefit you, and everyone else who uses GCC.

1.3 Programming in C and C++

C and C++ are languages that allow direct access to the computer's memory. Historically, they have been used for writing low-level systems soft-

[1] For details see the license file '`COPYING`' distributed with GCC.

ware, and applications where high-performance or control over resource usage are critical. However, great care is required to ensure that memory is accessed correctly, to avoid corrupting other data-structures. This book describes techniques that will help in detecting potential errors during compilation, but the risk in using languages like C or C++ can never be eliminated.

In addition to C and C++ the GNU Project also provides other high-level languages, such as GNU Common Lisp (`gcl`), GNU Smalltalk (`gst`), the GNU Scheme extension language (`guile`) and the GNU Compiler for Java (`gcj`). These languages do not allow the user to access memory directly, eliminating the possibility of memory access errors. They are a safer alternative to C and C++ for many applications.

1.4 Conventions used in this manual

This manual contains many examples which can be typed at the keyboard. A command entered at the terminal is shown like this,

```
$ command
```

followed by its output. For example:

```
$ echo "hello world"
hello world
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign '`$`' is used as the standard prompt in this manual, although some systems may use a different character.

When a command in an example is too long to fit in a single line it is wrapped and then indented on subsequent lines, like this:

```
$ echo "an example of a line which is too long to fit
    in this manual"
```

When entered at the keyboard, the entire command should be typed on a single line.

The example source files used in this manual can be downloaded from the publisher's website,[2] or entered by hand using any text editor, such as the standard GNU editor, `emacs`. The example compilation commands use `gcc` and `g++` as the names of the GNU C and C++ compilers, and `cc` to refer to other compilers. The example programs should work with any version of GCC. Any command-line options which are only available in recent versions of GCC are noted in the text.

The examples assume the use of a GNU operating system—there may be minor differences in the output on other systems. Some non-essential and verbose system-dependent output messages (such as very long system

---

[2] See `http://www.network-theory.co.uk/gcc/intro/`

paths) have been edited in the examples for brevity. The commands for setting environment variables use the syntax of the standard GNU shell (`bash`), and should work with any version of the Bourne shell.

2 Compiling a C program

This chapter describes how to compile C programs using `gcc`. Programs can be compiled from a single source file or from multiple source files, and may use system libraries and header files.

Compilation refers to the process of converting a program from the textual *source code*, in a programming language such as C or C++, into *machine code*, the sequence of 1's and 0's used to control the central processing unit (CPU) of the computer. This machine code is then stored in a file known as an *executable file*, sometimes referred to as a *binary file*.

2.1 Compiling a simple C program

The classic example program for the C language is *Hello World*. Here is the source code for our version of the program:

```
#include <stdio.h>

int
main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

We will assume that the source code is stored in a file called '`hello.c`'. To compile the file '`hello.c`' with `gcc`, use the following command:

```
$ gcc -Wall hello.c -o hello
```

This compiles the source code in '`hello.c`' to machine code and stores it in an executable file '`hello`'. The output file for the machine code is specified using the '`-o`' option. This option is usually given as the last argument on the command line. If it is omitted, the output is written to a default file called '`a.out`'.

Note that if a file with the same name as the executable file already exists in the current directory it will be overwritten.

The option '`-Wall`' turns on all the most commonly-used compiler warnings—**it is recommended that you always use this option!** There are many other warning options which will be discussed in later chapters, but '`-Wall`' is the most important. GCC will not produce any warnings unless they are enabled. Compiler warnings are an essential aid in detecting problems when programming in C and C++.

In this case, the compiler does not produce any warnings with the '-Wall' option, since the program is completely valid. Source code which does not produce any warnings is said to *compile cleanly*.

To run the program, type the path name of the executable like this:

```
$ ./hello
Hello, world!
```

This loads the executable file into memory and causes the CPU to begin executing the instructions contained within it. The path `./` refers to the current directory, so `./hello` loads and runs the executable file '`hello`' located in the current directory.

2.2 Finding errors in a simple program

As mentioned above, compiler warnings are an essential aid when programming in C and C++. To demonstrate this, the program below contains a subtle error: it uses the function `printf` incorrectly, by specifying a floating-point format '`%f`' for an integer value:

```
#include <stdio.h>

int
main (void)
{
  printf ("Two plus two is %f\n", 4);
  return 0;
}
```

This error is not obvious at first sight, but can be detected by the compiler if the warning option '`-Wall`' has been enabled.

Compiling the program above, '`bad.c`', with the warning option '`-Wall`' produces the following message:

```
$ gcc -Wall bad.c -o bad
bad.c: In function ‘main’:
bad.c:6: warning: double format, different
  type arg (arg 2)
```

This indicates that a format string has been used incorrectly in the file '`bad.c`' at line 6. The messages produced by GCC always have the form *file:line-number:message*. The compiler distinguishes between *error messages*, which prevent successful compilation, and *warning messages* which indicate possible problems (but do not stop the program from compiling).

In this case, the correct format specifier should be '`%d`' for an integer argument. The allowed format specifiers for `printf` can be found in any general book on C, such as the *GNU C Library Reference Manual* (see [Further reading], page 111).

Without the warning option '-Wall' the program appears to compile cleanly, but produces incorrect results:

```
$ gcc bad.c -o bad
$ ./bad
Two plus two is 2.585495     (incorrect output)
```

The incorrect format specifier causes the output to be corrupted, because the function `printf` is passed an integer instead of a floating-point number. Integers and floating-point numbers are stored in different formats in memory, and generally occupy different numbers of bytes, leading to a spurious result. The actual output shown above may differ, depending on the specific platform and environment.

Clearly, it is very dangerous to develop a program without checking for compiler warnings. If there are any functions which are not used correctly they can cause the program to crash or produce incorrect results. Turning on the compiler warning option '-Wall' will catch many of the commonest errors which occur in C programming.

2.3 Compiling multiple source files

A program can be split up into multiple files. This makes it easier to edit and understand, especially in the case of large programs—it also allows the individual parts to be compiled independently.

In the following example we will split up the program *Hello World* into three files: 'main.c', 'hello_fn.c' and the header file 'hello.h'. Here is the main program 'main.c':

```
#include "hello.h"

int
main (void)
{
  hello ("world");
  return 0;
}
```

The original call to the `printf` system function in the previous program 'hello.c' has been replaced by a call to a new external function `hello`, which we will define in a separate file 'hello_fn.c'.

The main program also includes the header file 'hello.h' which will contain the declaration of the function `hello`. The declaration is used to ensure that the types of the arguments and return value match up correctly between the function call and the function definition. We no longer need to include the system header file 'stdio.h' in 'main.c' to declare the function `printf`, since the file ‘`main.c`’ does not call `printf` directly.

The declaration in ‘`hello.h`’ is a single line specifying the prototype of the function `hello`:

```
void hello (const char * name);
```

The definition of the function `hello` itself is contained in the file ‘`hello_fn.c`’:

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

void
hello (const char * name)
{
  printf ("Hello, %s!\n", name);
}
```

This function prints the message “`Hello,` *name*`!`” using its argument as the value of *name*.

Incidentally, the difference between the two forms of the include statement `#include "`*`FILE`*`.h"` and `#include <`*`FILE`*`.h>` is that the former searches for ‘*`FILE`*`.h`’ in the current directory before looking in the system header file directories. The include statement `#include <`*`FILE`*`.h>` searches the system header files, but does not look in the current directory by default.

To compile these source files with `gcc`, use the following command:

```
$ gcc -Wall main.c hello_fn.c -o newhello
```

In this case, we use the ‘`-o`’ option to specify a different output file for the executable, ‘`newhello`’. Note that the header file ‘`hello.h`’ is not specified in the list of files on the command line. The directive `#include "hello.h"` in the source files instructs the compiler to include it automatically at the appropriate points.

To run the program, type the path name of the executable:

```
$ ./newhello
Hello, world!
```

All the parts of the program have been combined into a single executable file, which produces the same result as the executable created from the single source file used earlier.

2.4 Compiling files independently

If a program is stored in a single file then any change to an individual function requires the whole program to be recompiled to produce a new executable. The recompilation of large source files can be very time-consuming.

When programs are stored in independent source files, only the files which have changed need to be recompiled after the source code has been modified. In this approach, the source files are compiled separately and then *linked* together—a two stage process. In the first stage, a file is compiled without creating an executable. The result is referred to as an *object file*, and has the extension '`.o`' when using GCC.

In the second stage, the object files are merged together by a separate program called the *linker*. The linker combines all the object files to create a single executable.

An object file contains machine code where any references to the memory addresses of functions (or variables) in other files are left undefined. This allows source files to be compiled without direct reference to each other. The linker fills in these missing addresses when it produces the executable.

2.4.1 Creating object files from source files

The command-line option '`-c`' is used to compile a source file to an object file. For example, the following command will compile the source file '`main.c`' to an object file:

```
$ gcc -Wall -c main.c
```

This produces an object file '`main.o`' containing the machine code for the `main` function. It contains a reference to the external function `hello`, but the corresponding memory address is left undefined in the object file at this stage (it will be filled in later by linking).

The corresponding command for compiling the `hello` function in the source file '`hello_fn.c`' is:

```
$ gcc -Wall -c hello_fn.c
```

This produces the object file '`hello_fn.o`'.

Note that there is no need to use the option '`-o`' to specify the name of the output file in this case. When compiling with '`-c`' the compiler automatically creates an object file whose name is the same as the source file, but with '`.o`' instead of the original extension.

There is no need to put the header file '`hello.h`' on the command line, since it is automatically included by the `#include` statements in '`main.c`' and '`hello_fn.c`'.

2.4.2 Creating executables from object files

The final step in creating an executable file is to use gcc to link the object files together and fill in the missing addresses of external functions. To link object files together, they are simply listed on the command line:

```
$ gcc main.o hello_fn.o -o hello
```

This is one of the few occasions where there is no need to use the '-Wall' warning option, since the individual source files have already been successfully compiled to object code. Once the source files have been compiled, linking is an unambiguous process which either succeeds or fails (it fails only if there are references which cannot be resolved).

To perform the linking step gcc uses the linker ld, which is a separate program. On GNU systems the GNU linker, GNU ld, is used. Other systems may use the GNU linker with GCC, or may have their own linkers. The linker itself will be discussed later (see Chapter 11 [How the compiler works], page 89). By running the linker, gcc creates an executable file from the object files.

The resulting executable file can now be run:

```
$ ./hello
Hello, world!
```

It produces the same output as the version of the program using a single source file in the previous section.

2.5 Recompiling and relinking

To show how source files can be compiled independently we will edit the main program 'main.c' and modify it to print a greeting to everyone instead of world:

```
#include "hello.h"

int
main (void)
{
  hello ("everyone");  /* changed from "world" */
  return 0;
}
```

The updated file 'main.c' can now be recompiled with the following command:

```
$ gcc -Wall -c main.c
```

This produces a new object file 'main.o'. There is no need to create a new object file for 'hello_fn.c', since that file and the related files that it depends on, such as header files, have not changed.

The new object file can be relinked with the `hello` function to create a new executable file:

```
$ gcc main.o hello_fn.o -o hello
```

The resulting executable '`hello`' now uses the new `main` function to produce the following output:

```
$ ./hello
Hello, everyone!
```

Note that only the file '`main.c`' has been recompiled, and then relinked with the existing object file for the `hello` function. If the file '`hello_fn.c`' had been modified instead, we could have recompiled '`hello_fn.c`' to create a new object file '`hello_fn.o`' and relinked this with the existing file '`main.o`'.(1)

In a large project with many source files, recompiling only those that have been modified can make a significant saving. The process of recompiling only the modified files in a project can be automated with the standard Unix program `make`.

2.6 A simple makefile

For those unfamiliar with `make`, this section provides a simple demonstration of its use. Make is a program in its own right and can be found on all Unix systems. To learn more about the GNU version of `make` you will need to consult the *GNU Make* manual by Richard M. Stallman and Roland McGrath (see [Further reading], page 111).

Make reads a description of a project from a *makefile* (by default, called '`Makefile`' in the current directory). A makefile specifies a set of compilation rules in terms of *targets* (such as executables) and their *dependencies* (such as object files and source files) in the following format:

```
target: dependencies
        command
```

For each target, make checks the modification time of the corresponding dependency files to determine whether the target needs to be rebuilt using the corresponding command. Note that the *`command`* lines in a makefile must be indented with a single ⟨TAB⟩ character, not spaces.

GNU Make contains many default rules, referred to as *implicit* rules, to simplify the construction of makefiles. For example, these specify that '`.o`' files can be obtained from '`.c`' files by compilation, and that an executable can be made by linking together '`.o`' files. Implicit rules are defined in terms of *make variables*, such as `CC` (the C compiler) and `CFLAGS`

(1) If the prototype of a function has changed, it is necessary to modify and recompile all of the other source files which use it.

(the compilation options for C programs), which can be set using *VARIABLE=VALUE* lines in the makefile. For C++ the equivalent variables are `CXX` and `CXXFLAGS`, while the make variable `CPPFLAGS` sets the preprocessor options. The implicit and user-defined rules are automatically chained together as necessary by GNU Make.

A simple '`Makefile`' for the project above can be written as follows:

```
CC=gcc
CFLAGS=-Wall
main: main.o hello_fn.o

clean:
        rm -f main main.o hello_fn.o
```

The file can be read like this: using the C compiler `gcc`, with compilation option '`-Wall`', build the target executable `main` from the object files '`main.o`' and '`hello_fn.o`' (these, in turn, will be built via implicit rules from '`main.c`' and '`hello_fn.c`'). The target `clean` has no dependencies and simply removes all the compiled files.(2) The option '`-f`' (force) on the `rm` command suppresses any error messages if the files do not exist.

To use the makefile, type `make`. When called with no arguments, the first target in the makefile is built, producing the executable '`main`':

```
$ make
gcc -Wall   -c -o main.o main.c
gcc -Wall   -c -o hello_fn.o hello_fn.c
gcc   main.o hello_fn.o   -o main
$ ./main
Hello, world!
```

To rebuild the executable after modifying a source file, simply type `make` again. By checking the timestamps of the target and dependency files, make identifies the files which have changed and regenerates the corresponding intermediate files needed to update the targets:

```
$ emacs main.c   (edit the file)
$ make
gcc -Wall   -c -o main.o main.c
gcc    main.o hello_fn.o   -o main
$ ./main
Hello, everyone!
```

Finally, to remove the generated files, type `make clean`:

```
$ make clean
rm -f main main.o hello_fn.o
```

(2) This assumes that there is no file called '`clean`' in the current directory—see the discussion of "phony targets" in the GNU Make manual for details.

A more sophisticated makefile would usually contain additional targets for installation (`make install`) and testing (`make check`).

The examples in the rest of this book are small enough not to need makefiles, but the use of make is recommended for any larger programs.

2.7 Linking with external libraries

A library is a collection of precompiled object files which can be linked into programs. The most common use of libraries is to provide system functions, such as the square root function `sqrt` found in the C math library.

Libraries are typically stored in special *archive files* with the extension '`.a`', referred to as *static libraries*. They are created from object files with a separate tool, the GNU archiver `ar`, and used by the linker to resolve references to functions at compile-time. We will see later how to create libraries using the `ar` command (see Chapter 10 [Compiler-related tools], page 81). For simplicity, only static libraries are covered in this section—dynamic linking at runtime using *shared libraries* will be described in the next chapter.

The standard system libraries are usually found in the directories '`/usr/lib`' and '`/lib`'.(3) For example, the C math library is typically stored in the file '`/usr/lib/libm.a`' on Unix-like systems. The corresponding prototype declarations for the functions in this library are given in the header file '`/usr/include/math.h`'. The C standard library itself is stored in '`/usr/lib/libc.a`' and contains functions specified in the ANSI/ISO C standard, such as '`printf`'—this library is linked by default for every C program.

Here is an example program which makes a call to the external function `sqrt` in the math library '`libm.a`':

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

int
main (void)
{
  double x = sqrt (2.0);
  printf ("The square root of 2.0 is %f\n", x);
  return 0;
}
```

---

(3) On systems supporting both 64 and 32-bit executables the 64-bit versions of the libraries will often be stored in '`/usr/lib64`' and '`/lib64`', with the 32-bit versions in '`/usr/lib`' and '`/lib`'.

Trying to create an executable from this source file alone causes the compiler to give an error at the link stage:

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function `main':
/tmp/ccbR6Ojm.o(.text+0x19): undefined reference
  to `sqrt'
```

The problem is that the reference to the `sqrt` function cannot be resolved without the external math library '`libm.a`'. The function `sqrt` is not defined in the program or the default library '`libc.a`', and the compiler does not link to the file '`libm.a`' unless it is explicitly selected. Incidentally, the file mentioned in the error message '`/tmp/ccbR60jm.o`' is a temporary object file created by the compiler from '`calc.c`', in order to carry out the linking process.

To enable the compiler to link the `sqrt` function to the main program '`calc.c`' we need to supply the library '`libm.a`'. One obvious but cumbersome way to do this is to specify it explicitly on the command line:

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

The library '`libm.a`' contains object files for all the mathematical functions, such as `sin`, `cos`, `exp`, `log` and `sqrt`. The linker searches through these to find the object file containing the `sqrt` function.

Once the object file for the `sqrt` function has been found, the main program can be linked and a complete executable produced:

```
$ ./calc
The square root of 2.0 is 1.414214
```

The executable file includes the machine code for the main function and the machine code for the `sqrt` function, copied from the corresponding object file in the library '`libm.a`'.

To avoid the need to specify long paths on the command line, the compiler provides a short-cut option '`-l`' for linking against libraries. For example, the following command,

```
$ gcc -Wall calc.c -lm -o calc
```

is equivalent to the original command above using the full library name '`/usr/lib/libm.a`'.

In general, the compiler option '`-l`*NAME*' will attempt to link object files with a library file '`lib`*NAME*`.a`' in the standard library directories. Additional directories can specified with command-line options and environment variables, to be discussed shortly. A large program will typically use many '`-l`' options to link libraries such as the math library, graphics libraries and networking libraries.

2.7.1 Link order of libraries

The traditional behavior of linkers is to search for external functions from left to right in the libraries specified on the command line. This means that a library containing the definition of a function should appear after any source files or object files which use it. This includes libraries specified with the short-cut '-l' option, as shown in the following command:

```
$ gcc -Wall calc.c -lm -o calc    (correct order)
```

With some linkers the opposite ordering (placing the '-lm' option before the file which uses it) would result in an error,

```
$ cc -Wall -lm calc.c -o calc    (incorrect order)
main.o: In function 'main':
main.o(.text+0xf): undefined reference to 'sqrt'
```

because there is no library or object file containing sqrt after 'calc.c'. The option '-lm' should appear after the file 'calc.c'.

When several libraries are being used, the same convention should be followed for the libraries themselves. A library which calls an external function defined in another library should appear before the library containing the function.

For example, a program 'data.c' using the GNU Linear Programming library 'libglpk.a', which in turn uses the math library 'libm.a', should be compiled as,

```
$ gcc -Wall data.c -lglpk -lm
```

since the object files in 'libglpk.a' use functions defined in 'libm.a'.

Most current linkers will search all libraries, regardless of order, but since some do not do this it is best to follow the convention of ordering libraries from left to right.

This is worth keeping in mind if you ever encounter unexpected problems with undefined references, and all the necessary libraries appear to be present on the command line.

2.8 Using library header files

When using a library it is essential to include the appropriate header files, in order to declare the function arguments and return values with the correct types. Without declarations, the arguments of a function can be passed with the wrong type, causing corrupted results.

The following example shows another program which makes a function call to the C math library. In this case, the function pow is used to compute the cube of two (2 raised to the power of 3):

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

```
int
main (void)
{
  double x = pow (2.0, 3.0);
  printf ("Two cubed is %f\n", x);
  return 0;
}
```

However, the program contains an error—the `#include` statement for '`math.h`' is missing, so the prototype `double pow (double x, double y)` given there will not be seen by the compiler.

Compiling the program without any warning options will produce an executable file which gives incorrect results:

```
$ gcc badpow.c -lm
$ ./a.out
Two cubed is 2.851120      (incorrect result, should be 8)
```

The results are corrupted because the arguments and return value of the call to `pow` are passed with incorrect types.(4) This can be detected by turning on the warning option '`-Wall`':

```
$ gcc -Wall badpow.c -lm
badpow.c: In function 'main':
badpow.c:6: warning: implicit declaration of
  function 'pow'
```

This example shows again the importance of using the warning option '`-Wall`' to detect serious problems that could otherwise easily be overlooked.

(4) The actual output shown above may differ, depending on the specific platform and environment.

Index

#

$

-

B

C

D

E

F

G

J

K

L

M

N

O

P

Q

R