Learning GNU C

Ciaran O'Riordan

Learning GNU C
by Ciaran O'Riordan

This file is a C programming tutorial using the GNU C compiler and GNU Libc.

Table of Contents

List of Examples

Preface

1. Target Audience

Welcome to *Learning GNU C*. The aim of this book is to teach GNU users how to write software in C. It is written primarily as a tutorial for beginners but should be thorough enough to be used as a reference by experience programmers. The basics are layed down in full in the first few chapters, beginners will read these chapters carefully while those with prior experience can skim through them. All the information is there, no prior knowledge of programming is assumed.

The reader is assumed to have access to a computer with a GNU system installed. Although the target audience is GNU users, the content of the book should also be 98% relevant to users of OpenBSD, FreeBSD, or NetBSD. Some familiarity with using your computer from the shell (the command line) would be helpful although all commands will be shown along side programming examples. The only piece of software you do need experience with is a text editor. Any text editor will do. GNU Emacs is an especially good one for programmers. It has been in development for over twenty years and contains hundreds useful features. GNU Nano is a simple text editor you could use, some programmers like to use vi (pronounced vee eye). If you already have a favorite text editor, you can use that. There are also graphical editors geared towards programmers such as Anjuta and KDevelop but most programmers preferr text based editors.

2. Scope of this text

The contents of this book can be divided into two topics: the core C language, and the standard functionality made available to the programmer. The standard functionality I mention is provided by *GNU Libc*, this is a *library* of C functionality that is part of every GNU system. Neither of these topics is of much use without the other but there is a focus on the core langauge near the beginning and more discussion on Libc near the end. The ordering of topics is designed to teach C programming in an incremental fashion where each chapter builds on the previous one. Some aspects of the core langauge are only really of use to experienced programmers and so appear near the end.

The C language on it’s own can make decisions, repeat commands, store data, and perform mathematics. Equally importantly, it provides a method to make use of extra functionality such as Libc.

Libc provides functionality such as reading and writing files, sorting and searching data, getting input from the user, displaying data to the user, communicating across networks, creating easily translatable programs, and many other things.

3. Why learn C?

C is a standard. It is the programmers programming language. It is the standard programming language of GNU and BSD based systems. The majority of these systems and the applications that run on them, is written in C. C was developed over thirty years ago for writing operating systems and applications. It's small, extensible design has allowed it to evolve with the computer industry. Because of it's age and popularity, C is a very well supported language. Many tools exist to make C programming easier and these tools are often very mature and of a high standard. All the software we will use in this book is written in C.

4. Why use GNU?

GNU is a complete, Unix-like operating system that has been in development for just over twenty years. GNU software is known for it's stability and standard compliance.

Most GNU systems use *Linux* as a kernel. These systems are often known as GNU/Linux systems.

5. Why Free Software?

The greatest thing about GNU is that the entire system is what is known as *Free Software*. Software is "Free Software" when people have the freedom to: Use the software for any purpose, make changes to the software, share the software with others, and distribute modified versions of the software.

Software that isn't Free Software is called *proprietary* software. It is so-called because a person claims the software to be their property and bans others from sharing and making changes to it.

From an ethical standpoint, writing Free Software is a much more social way to act. Free Software empowers it's users by allowing them to help themselves by making changes they want to the software (or getting someone else to make these changes). It allows people to help their neighbours by sharing the software, proprietary software does the opposite: it makes sharing illegal telling people that it is a criminal offenses to say "yes" when someone asks for help. And Free Software allows people to help their community by distributing improved versions of the software.

Free Software also doesn't descriminate against poorer people or people from developing nations. By allowing them all the above freedoms it permits them to use computers without having to pay impossible amounts of money for the "privilage".

Finally there are the technical benefits. Free Software is free from marketing plots. It doesn't restrict itself to force users to buy extra pieces of software. Each piece of GNU is designed to be as useful as possible. As a programmer you can use the same C programming software that is used in major projects.

Non-Free programs are generally distributed in a machine readable form only. This means that the user cannot see what is going on inside a program. In contrast, Free Software is required to come with source code in a human readable format. As a programmer you can read the source code for any piece os Free Software you like. If there are errors in a program, you can fix them.

This freedom to fix errors and add functionality is what has made GNU software so good. All code is available for peer review.

Free Software will change the world for the better

6. References to persons

The terms “he” and “she” will be used as little as possible. They are usually unnecessary. The reader will be referred to as “the reader” or “you” while imaginary people will be called by there role (i.e. “the programmer” or “the manager”. When referring to myself I will use “we”. This is so that the text will not have to be changed if others help out with writing the book. (NOTE: I’m not looking for co-authors right now.) If I get stuck and have to use “he” or “she” I will opt for the latter. This is to balance up the bias of most books. I will also ask the reader to picture this female to have two disabilities and be of an under-represented race (i.e. an blind, stuttering eskimo). Thank you.

Chapter 1. Introduction to C

1.1. What are Programming Languages?

A programming langauge defines a format for laying out ordered sets of intructions to be executed by a computer. Programming langauges can be sorted into three three categories: interpretted languages, compiled languages, and machine languages. Of these types only machine languages can be understood directly by a computer.

A machine langauge is the set of instructions that a computer's CPU (central processing unit) understands. All instructions and information are represented by numbers; very fast for computers, very hard for human brains to read or write. To ease the task of computer programming, people created easier languages called *assembly languages*. An assembly language is one which provides textual names for the available machine language commands. This, along with the fact that assembly languages allowed programmers to add spaces and tabs to their code, made assembly languages far easier to program with. Assembly code can then be fed to an *assembler* which translates it into the machine language of the target computer's CPU.

The use of assembly languages spread very fast, they became known as "second generation languages" but there was still two problems with assembly langauges. Firstly, each command does only a very basic task such as add two numbers or load a value from memory. Using these small commands was quite tedious. The second problem was much bigger. Programs written in an assembly language are bound to a particular type of CPU. Each type of CPU has it's own machine language and, therefore, it's own assembly language. The next task was to design a language that could be translated into the machine language of many CPUs.

These new *machine independant* langauges were known as "third generation" or "high-level" languages. Designed to be easy to read, these languages were made up of english words, basic mathematical symbols and a few punctuation characters. These languages allow simple statements to be expressioned concisely, for example, adding two numbers and storing the result in memory could be expressed as:

```
data = 10 + 200;
```

rather than:

```
Load R1, 10
Load R2, 200
Addi R1, R2
Store R2, L1
```

1.2. What is C?

A tool called a *compiler* is then used to convert the high-level code into machine language. A program can be written in C and compiled for any computer, it's up to the compiler to get the hardware-specific instructions right.

To see just how readable C is compared to Assembly language, take a look at the following tiny program written in each:

Example 1-1. C vs. Assembly language

```
          .section  .rodata
.LC0:
          .string   "Tax Due: %d\n"
          .text
          .align 2
.globl main
          .type     main,@function
main:
          pushl     %ebp
          movl      %esp, %ebp
          subl      $24, %esp
          andl      $-16, %esp
          movl      $0, %eax
          subl      %eax, %esp
          movl      $1000, %eax
          movl      $400, %edx
          movl      $0x3e6147ae, -12(%ebp)
          subl      %edx, %eax
          pushl     %eax
          fildl     (%esp)
          leal      4(%esp), %esp
          fmuls     -12(%ebp)
          fnstcw    -18(%ebp)
          movw      -18(%ebp), %ax
          movb      $12, %ah
          movw      %ax, -20(%ebp)
          fldcw     -20(%ebp)
          fistpl    -16(%ebp)
          fldcw     -18(%ebp)
          subl      $8, %esp
          pushl     -16(%ebp)
          pushl     $.LC0
          call      printf
          addl      $16, %esp
          movl      $1, %eax
          leave
          ret
.Lfe1:
          .size     main,.Lfe1-main
```

And the program in C:

```
#include <stdio.h>

int
main()
{
  int wages = 1000;
  int tax_allowance = 400;
  float tax_rate = 0.22;
  int tax_due;

  tax_due = (wages - tax_allowance) * tax_rate;

  printf("Tax Due: %d euro\n", tax_due);

  return 0;
}
```

Which did you find easier to understand, even without knowing C. The output of both programs is the same: “Tax Due: 131 euro”. The Assembly code shown is written in the “80386” instruction set, it will not work on machines that use a different instruction set. The C code can be compiled for practically any computer.

1.3. Programming Tools

GNU comes with a compiler called *GCC*. Originally this stood for “GNU C Compiler” but since it can now compile languages other than C it’s name was changed to “GNU Compiler Collection”. To check if you have GCC installed, type the following:

```
ciaran@pooh:~/book$ gcc --version
3.2.1
ciaran@pooh:~/book$
```

The version of GCC you have installed may be different, anything similar, such as “2.95.2” or “3.3.0”, is ok. If you got an error message saying **command not found** then you don’t have GCC installed. If you installed GNU from a CD, you should find GCC there. If you don’t know how to install applications from a CD then get a friend or the person who installed your GNU system to do it for you.

1.4. Introducing GCC

Now we’re going to show you a tiny bit of C code and how to compile it. The point here is to show you how to use GCC so we won’t explain the C code yet. Here’s the smallest C program that GCC will compile. It does nothing.

Example 1-2. tiny.c

```
main()
{
}
```

Type this piece of code into your text editor and save it to a file called `tiny.c`. You can choose any name so long as it ends with `.c`, this is the extension used by C programs, GCC checks for this extension when compiling a program. With the file saved, you can now compile it into an executable program by typing:

```
ciaran@pooh:~/book$ gcc tiny.c
ciaran@pooh:~/book$
```

This command should succeed with no output. If you got any error messages check that you typed the program in correctly. Weighing in at eight characters we'll assume you've gotten this much correct and move on. A file called `a.out` should have appeared in your directory. This is the machine language program created from the above code, if you run it you will see that it really does nothing at all. The name `a.out` exists for historical reasons, it is short for *assembler output*.

Although GCC will compile this code, it isn't strictly complete. If we enable GCC's warnings we will be told what is missing. You are not expected to understand the warning messages right now, we show this only to demonstrate GCC's warnings. You enable warnings by adding the **-Wall** switch to your compilation command.

```
ciaran@pooh:~/book$ gcc -Wall tiny.c
tiny.c:2: warning: return-type defaults to ‘int’
tiny.c: In function ‘main’:
tiny.c:3: warning: control reaches end of non-void function
ciaran@pooh:~/book$
```

These warnings appear because our program is not strictly complete. To get rid of these warnings we must add two more lines. So here's the smallest *valid* C program.

Example 1-3. tiny2.c

```
int
main()
{
  return 0;
}
```

When we compile this with the **-Wall** option we will get no warnings. Another option: **-o filename** can be used to specify the name you want to give to your program (instead of `a.out`).

```
ciaran@pooh:~/book$ gcc -Wall -o tiny-program tiny2.c
ciaran@pooh:~/book$ ls
```

```
tiny2.c  tiny-program
ciaran@pooh:~/book$ ./tiny-program
ciaran@pooh:~/book$
```

1.5. Who defines *Valid* C?

For you, the programmer, “valid C” is defined by the compiler. There are many dialects of C in existence, thankfully they are all very similar. There are also other languages that are based on C such as *Objective C* and *C++*. These languages are very like C in there appearance but there usages is quite different. GCC understands many dialects of C as well as many other languages (including Objective C and C++).

1.5.1. “K&R” C

C was created by Dennis Ritchie between 1969 and 1973. In 1978 Dennis Ritchie along with Brian Kernighan published an excellent C tutorial “The C programming language”. This was the first formal definition of the language. Being the original C dialect it is sometimes called *Traditional C*. Unfortunately the book left many aspects of the language undefined, this meant that people writing compilers had to make decisions as to how to handle these aspects. The result was that a piece of code would behave differently depending on what compiler was used. This dialect is no longer used, GCC supports it only for compiling very old programs. We mention it here purely for historical purposes.

1.5.2. ISO C

In 1983 the American National Standards Institute (*ANSI*) set up a committee to draw up a more exact standard and fix a few shortcomings they saw in the language. In 1989 they finalised this standard which was accepted by the International Standards Organisation (*ISO*). This new dialect became known as “C89”. It is also called “ISO C” or “ANSI C”. GCC is one of the most conforming compilers available.

1.5.3. C99

The ANSI C committee meets infrequently to update the standard. The latest updated standard was released in 1999 and is known as “C99”. Few compilers fully support C99 yet; making changes to one of the most important pieces of software to an operating system takes time. GCC’s C99 support is mostly complete (at the time of this writing) but the developers are working on it.

1.5.4. GNU C

GNU C is most similar to C89 but has a lot of the new features of C99 added and a few other extensions. These extensions have been added conservatively by the developers as problems are found that C99 doesn't provide good solutions to. GNU C is the default dialect of GCC and is the dialect we will use in this book. We will try our best to point out GNU extensions when we use them but in general, it is better to make full use GNU C. Use of ISO C is limiting your programs to the lowest common denominator and should only be used in special cases.

1.5.5. Choosing a Dialect

If you would like to use a dialect other than the default, you can specify your choice with the **-std=** switch followed by name of the dialect. The names are: **c89**, **c99**, **gnu89** and, **gnu99**. "gnu89" is the current default but "gnu99" will become the default when C99 support is complete. The change will not be very noticeable.

1.5.6. Future Standards

Extensions such as those added by GCC are the main source of inspiration for new ISO C standards. When the ANSI C group see a lot of compilers implementing an extension they review the necessity of that feature and if they decide it would be of benefit they work out a standard way to implement it. Some of GCC's extensions may make it into the next standard, some will not.

1.6. Conclusion

This concludes our introduction. Hopefully you now have a grasp of what programming is. In the next chapter we'll start writing basic programs that actually do something and explain how they do it.

Chapter 2. Staring With Functions

2.1. What are functions?

Functions are the building blocks of C programs. The majority of a C program is made up of named blocks of code called *functions*. When you write a program you will write many functions to perform the tasks you need. There are, however, a lot of common tasks such as displaying text to the screen that a lot of programmers will need. Instead of having everyone reinventing the wheel, GNU systems come with *libraries* of pre-defined functions for many of these tasks. Over the years, thousands of such functions have accumulated. If you were writing a program that plays the game, BINGO, you would have to write the game specific functions yourself but you would find that others have already written functions for generating random numbers, displaying results to the screen, getting input from the player etc.

Every C program must have a function called **main()**, this is where execution of the program begins. The code of a program *could* be completely contained in **main()** but it is more usual to split a program into many small functions.

The first piece of useful code we will look at is a classic. When compiled and run it will display a simple greeting to your screen. This program *defines* a function called **main()** and *calls* (uses) a function called **printf()**. **printf()** is a function provided for us by the "Standard Device Input/Output library". This library comes with every GNU system. Here's our little program:

Example 2-1. hello.c

```
#include <stdio.h>

int
main()
{
  printf("hello, world\n");

  return 0;
}
```

Compile and run this program before moving on. If all goes well, it will display the text string "hello, world" to your terminal (the standard output device). Here's the compilation command just in case you've forgotten:

```
ciaran@pooh:~/book$ gcc -Wall -o hello hello.c
ciaran@pooh:~/book$ ./hello
hello, world
ciaran@pooh:~/book$
```

If you got any error or warning messages check that your code matches the code in this book exactly. Any messages you got should tell you the line of code where your mistake is. If you've typed the code in correctly you will get no such messages.

2.2. A Line-by-Line Dissection

We'll do a quick description of what each line does. Don't worry if your not sure about some parts, we'll do plenty more examples.

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

This line tells GCC to *include* information about how to use the functions from the Standard Device Input/Output library. Usually the standard input device is your keyboard and the standard output device a terminal (which is displayed on your monitor. This library is very widely used, we'll come across a lot of functions from it in this book.

```
int
main()
```

These two lines begin the definition of the function **main()**. We'll explain the first of these two lines later.

```
{
```

The open curly braces signals the beginning of a block of code. All code between this curly brace and it's matching closing brace is part of the function **main()**.

```
  printf("hello, world\n");
```

This line is a *function call*, the function is already defined for you. When you call **printf()** you must pass it an *argument* to tell it what to display.

```
  return 0;
```

The **return** statement ends execution of the function **main()**, any statements after this line would not be executed. When **main()** ends your program exits. When a function ends, it can pass a value back to whoever called it, this is done by placing the value after **return**. **main()** always *returns* an integer (a positive or negative number with no decimal point). We tell the compiler to expect this by preceding the

definition of **main()** with **int**. When returning from **main()** it is convention to return zero if no problems were encountered.

```
}
```

The closing curly brace signals the end of the block of code that makes up **main()**.

The two lines that make up the body of **main()** are known as *statements*. More specifically they are *simple statements* (as opposed to compound statements which we will encounter in chapter 4). Statements are to C what sentences are to spoken languages. A semi-colon ends a simple statement. The blank lines in the program are optional, C never requires a blank line but they make code much easier to read.

We mentioned that our function **main()** *returns* the value zero. For most functions the return value can be used within the program but since returning from **main()** signals the end of the program it returns it to the shell. The return value of a program is stored by the shell, if you want to see it, type the following:

```
ciaran@pooh:~/book$ gcc -Wall -o hello hello.c
ciaran@pooh:~/book$ ./hello
hello, world
ciaran@pooh:~/book$ echo $?
0
ciaran@pooh:~/book$
```

2.3. Comments

Comments are a way to add explanatory text to your program, they are ignored by the compiler so they don't affect your program in any way. As the programs you write get larger you will find it helpful to have comments in your code to remind you what you are doing. In the examples in this book we will use comments to explain what is going on. There are two ways to insert a comment into your program, the most common way is to start and end your comments with **/*** and ***/** respectively. Comments of this sort can span multiple lines. The second way is by placing **//** at the start of your comment. Comments of this sort are terminated at the end of a line. Here's our "hello, world" program with comments.

Example 2-2. hello2.c

```
/* The purpose of this program is to
 * display some text to the screen
 * and then exit.
 */

#include <stdio.h>
```

```
int
main()
{
  /* printf() displays a text string */
  printf("hello, world\n");

  return 0;  //zero indicates there were no errors
}
```

When compiled, this code will produce exactly the same executable. Lines 2 and 3 of the comment at the top start with an asterisk, this is not necessary but it makes it clear that the comment extends for four lines.

2.4. Making your own Functions

In that last example we defined just one function. To add another function you must generally do two things. First you must *define* the function, just like we defined **main()**. Also you you must *declare* it. Declaring a function is like telling GCC to expect it, we didn't have to declare **main()** because it is a special function and GCC knows to expect it. The name, or identifier, you give to a function must appear in both the definition and the declaration.

Functions identifiers can be made up of the alphabetic characters "a"-"z" and "A"-"Z", the numeric characters "0"-"9" and the underscore character "_". These can be used in any order so long as the first character of the identifier is not a number. As we said earlier, C is case-sensitive so **My_Function** is completely different to **my_function**. A functions identifier must be unique. Identifiers can safely be up to 63 characters long or as short as 1 character.

Along with it's identifier you must give each function a *type* and a block of code. The *type* tells the compiler what sort of data it *returns*. The return value of a function can be ignored, **printf()** returns an integer saying how many character it displayed to the terminal. This information wasn't important to us so we ignored it in our program. In the next chapter we'll discuss types of data in detail, until then we'll gloss over return values.

Here's a program that defines three functions:

Example 2-3. three_functions.c

```
#include <stdio.h>

/* function declarations */
int first_function(void);
int goodbye(void);


int
```

```
main()            // function definition
{
  printf("the program begins...\n");
  first_function();
  goodbye();

  return 0;
}


int
first_function()  // function definition
{
  /* this function does nothing */
  return 0;
}


int
goodbye()         // function definition
{
  printf("...and the program ends.\n");

  return 0;
}
```

In the above example we wrote **first_function()** which does nothing and **goodbye()** which displays a message. Functions must be declared *before* they can be called are called, in our case this means they must appear our definition of **main()**. In practice, function declarations are generally grouped at the top of a file after any **#include** lines and before any function definitions.

2.5. Multiple Files

Programs do not have to be written in just one file, your code can split up into as many files as you want, if a program is comprised of forty functions you *could* put each function into a separate file. This is a bit extreme though. Often functions are grouped by topic and put into separate files. Say you were writing a program that worked out the price of a pizza and displayed the result, you could put the calculation functions into one file, the display functions into another and have **main()** in a third one. The command you would use to compile your program would look something like this:

```
ciaran@pooh:~/book$ gcc -o pizza_program main.c prices.c display.c
```

Remember: If you define a function in `prices.c` and you want to *call* this function in `main.c` you must declare the function in `main.c`.

2.6. Header Files

Keeping track of function declarations can get messy, for this reason *Header files* are used to house C code that you wish to appear in multiple files. You have actually already used a header file. `stdio.h` is a header file which contains many function declarations, it contains the function declarations for **printf()** and **printf()**. Once you have placed the function declarations you wish to share into a header file you can **#include** your header in each C file that needs the information. The only difference being that you surround your filename in quotes instead of angle brackets (**"my_header.h"** instead of <**system_header.h**>). To illustrate these points we'll write that pizza program I mentioned earlier.

2.7. A Larger (non)Program

The small amount of programming we have shown so far isn't enough to make a decent interactive program. To keep it simple, we will write just a skeleton program so you can see the structure and usage of header files without getting bogged down in new concepts. In Chapter 3 we will write a full version of this program. The code here can be compiled and run but it will not ask the user for any input or calculate the price.

First we have `main.c`, this will only contain the function **main()**. **main()** will call some of the functions we define in other files. Note that `main.c` doesn't have a line **#include** <**stdio.h**> as it does not use any of the functions in the Standard Device I/O library.

Example 2-4. main.c

```
#include "display.h"
#include "prices.h"

int
main()
{
  display_options();
  calculate_price();
  display_price();

  return 0;
}
```

Next we have `display.c`. This contains two functions, both of which are called from **main()** and so we put there declarations in a header file `display.h`.

Example 2-5. display.c

```
#include <stdio.h>

int
display_options()
```

```
{
  printf("Welcome to the pizza parlor\n");
  printf("What size pizza would you like? (in inches)");

  return 0;
}

int
display_price()
{
  printf("Your pizza will cost 0.00\n");

  return 0;
}
```

Example 2-6. display.h

```
/* header file just contains function declarations, an file that wants
 * to use either of these functions just has to #include this file */
int display_options(void);
int display_price(void);
```

Finally we have `prices.c` which contains the functions for getting input from the user and calculating the total cost of the pizza. Only one of these functions is called from **main()**, the declarations for the other two are therefore put at the top of the file. We'll fill in the code for these functions in Chapter 3.

Example 2-7. prices.c

```
int get_size(void);
int get_toppings(void);

int
calculate_price()
{
  /* insert code here.  Will call get_size() and get_toppings(). */
  return 0;
}

int
get_size()
{
  /* insert code here */
  return 0;
}

int get_toppings()
{
  /* insert code here */
  return 0;
```

```
}
```

Example 2-8. prices.h

```
int calculate_price(void);
```

This can then be compiled with the command: **gcc -Wall -o pizza_program main.c prices.c display.c**. When run, it will display a greeting and announce that your pizza costs “£0.00”.

2.8. Another new Function

Before we move on, let’s take a look at one more function from the Standard Device I/O Library: **printf()**. The “Print Formatted” command is an advanced form of **printf()**. The string you pass to **printf()** can contain character sequences which have special meanings. Unlike **printf()**, there is no automatic new-line at the end of a string displayed by **printf()** to insert a new-line you add the characters **\n**.

2.9. Primer Summary

What we’ve covered so far shouldn’t be too hard. If you’d like to experiment, try writing similar programs that output a few lines. Split your program into a couple of functions and divide them into two files.

Always enable GCC’s warnings when compiling your program. Warnings mean your code is unclear or incomplete, GCC will guess at the correct meaning and will usually get it right but you should not rely on this. Looking at and correcting the warnings will help you get used to the language. Most warnings are accompanied by the line number where the problem is. If you can’t see anything wrong with that line check the line above it; if a statement is incomplete GCC won’t notice that it is an error until it encounters the beginning of the following statement. Don’t forget your semi-colons.

Chapter 3. Data and Expressions

Really useful programs take in data, perform actions on it and output it somewhere. In C, you use named pieces of memory called *variables* to store data. C programs can change the data stored in a variable at any time, hence the name. Every variable has an identifier which you can use to refer to it's data when you want to use or change it's value. An *expression* is anything that can be evaluated i.e. **1 + 1** is an expression of the value **2**. In this expression, the plus sign is a *binary operator*; it operates on two values to create a single value.

The rules for naming a variable are the same as for naming a function, you can use letters, numbers, and the underscore character and the first character must not be a number. Also like functions, variables must be declared before they can be used. The identifier you give to a variable should say what the the variable will be used for, this makes you code much easier to read. You can define your own variables or you can use one of the *types* already defined for you. Before we get bogged down in terminology let's look at a quick code example to show how simple it all is. In this example we will use two variables of the pre-defined type **int**.

Example 3-1. bicycles.c

```
#include <stdio.h>

int
main()
{
  int number_of_bicycles;
  int number_of_wheels;

  number_of_bicycles = 6;
  number_of_wheels = number_of_bicycles * 2;

  printf("I have %d bicycles\n", number_of_bicycles);
  printf("So I have %d wheels\n", number_of_wheels);

  return 0;
}
```

3.1. Bicycle Dissection

There are a few new things to look at here, we'll break the program into chunks to explain them.

```
  int number_of_bicycles;
  int number_of_wheels;
```

These two lines each declare a variable. **int** is one of the built-in data types of the C language. Variables of type **int** can store positive or negative whole numbers.

```
number_of_bicycles = 6;
```

This line stores the value **6** in the variable **number_of_bicycles**. The equals sign is known as “the assignment operator”, it assigns the value on the right hand side of it to the variable on the left hand side.

```
number_of_wheels = number_of_bicycles * 2;
```

Again, this line uses the assignment operator but it also uses the multiplication operator. The asterisk is another binary operator, it multiplies two values to create a single value. In this case it creates the value **12** which is then stored in **number_of_wheels**.

```
printf("I have %d bicycles\n", number_of_bicycles);
printf("So I have %d wheels\n", number_of_wheels);
```

Here we see **printf()** again but it’s being used unlike we have seen before. Here it is taking two *arguments* which are separated by a comma. The first argument to **printf()** is known as the *format string*. When a **%d** is encountered in the format string **printf()** knows to expect an extra argument. The **%d** is replaced by the value of this extra argument. One addition argument is expected for each **%d** encountered.

With this new knowledge it should be no surprise that when we compile and run this piece of code we get the following:

```
I have 6 bicycles
So I have 12 wheels
```

As always, don’t worry if you are unsure about certain parts. We’ll do plenty more examples.

3.2. Data Types

All the data types defined by C are made up of units of memory called *bytes*. On most computer architectures a byte is made up of eight *bits*, each bit stores a one or a zero. These eight bits with two states give 256 combinations (2^8). So an integer which takes up two bytes can store a number between 0 and 65535 (0 and 2^{16}. Usually however, integer variables use the first *bit* to store whether the number is positive or negative so their value will be between -32768 and +32767.

As we mentioned, there are eight basic data types defined in the C language. Five types for storing integers of varying sizes and three types for storing *floating point* values (values with a decimal point). C doesn't provide a basic data type for text. Text is made up of individual characters and characters are represented by numbers. In the last example we used one of the integer types: **int**. This is the most commonly used type in the C language.

The majority of data used in computer programs is made up of the integer types, we'll discuss the floating point types a little later. In order of size, starting with the smallest, the integer types are **char**, **short**, **int**, **long** and **long long**. The smaller types have the advantage of taking up less memory, the larger types incur a performance penalty. Variables of type **int** store the largest possible integer which does not incur this performance penalty. For this reason, **int** variables can be different depending what type of computer you are using.

The **char** data type is usually one byte, it is so called because they are commonly used to store single characters. The size of the other types is dependent on the hardware of your computer. Most desktop machines are "32-bit", this refers to the size of data that they are designed for processing. On "32-bit" machines the **int** data type takes up 4 bytes (2^{32}). The **short** is usually smaller, the **long** can be larger or the same size as an **int** and finally the **long long** is for handling very large numbers.

The type of variable you use generally doesn't have a big impact on the speed or memory usage of your application. Unless you have a special need you can just use **int** variables. We will try to point out the few cases where it can be important in this book. A decade ago, most machines had 16-bit processors, this limited the size of **int** variables to 2 bytes. At the time, **short** variables were usually also 2 bytes and **long** would be 4 bytes. Nowadays, with 32-bit machines, the default type (**int**) is usually large enough to satisfy what used to require a variable of type **long**. The **long long** type was introduced more recently to handle very large numeric values.

Some computers are better at handling really big numbers so the size of the data types will be bigger on these machines. To find out the size of each data type on your machine compile and run this piece of code. It uses one new language construct **sizeof()**. This tells you how many bytes a data type takes up.

Example 3-2. sizeof_types.c

```
int
main()
{
  printf("sizeof(char) == %d\n", sizeof(char));
  printf("sizeof(short) == %d\n", sizeof(short));
  printf("sizeof(int) == %d\n", sizeof(int));
  printf("sizeof(long) == %d\n", sizeof(long));
  printf("sizeof(long long) == %d\n", sizeof(long long));

  return 0;
}
```

3.3. Another Example of Assignment

Time for another example. This bit of code demonstrates a few more new things which we'll explain in a minute.

Example 3-3. displaying_variables.c

```
#include <stdio.h>

int
main()
{
  short first_number = -5;
  long second_number, third_number;

  second_number = 20000 + 10000;

  printf("the value of first_number is %hd\n", first_number);
  printf("the value of second_number is %ld\n", second_number);
  printf("the value of third_number is %ld\n", third_number);

  return 0;
}
```

We've used a **short** and two **long** variables. We could have used **int** variables but chose to use other types to show how similar they are. In the first line of **main()** we declare a variable and give it a value all in one line. This is pretty normal. The second line declares two variables at once by separating them with a comma. This can be handily but code is often more readable when variable declarations get a line to themselves.

The third line is very like some code from the first example, the addition operator produces the value **30000** which gets stored in **second_number**. The last thing to point out is that instead of **%d**, the format string of **printf()** contains **%hd** for the **short** variable and **%ld** for the long variables. These little groupings of characters are called *conversion specifiers*. Each type of variable has it's own conversion specifier. If you want to print a single percent sign ("%") you must write **%%**.

When you compile and run this you will see the value of your variables. The value of **third_number** will be strange. This is because it was never assigned a value. When you declare a variable, the operating system allocates some memory for it. You have no way of know what this memory was used for previously. Until you give your variable a value, the data stored in it is essentially random. Forgetting to assign a value to a variable is a common mistake among beginning programmers.

3.4. Quick Explanation of printf()

You may have noticed two characters near the end of our **printf()** statements **\n**. These don't get displayed to the screen, they are the notation **printf()** uses to represent "newline". **'\'** is the c *escape character* when it is encountered within quotes the following character usually has a special meaning. Another example is **\t** which is used to represent a TAB.

Another special character that **printf()** looks out for is **'%'**, this tells it to look at the next few characters and be ready to replace them with the value of a variable. **%d** is the character sequence that represents a variable of type **int** to be displayed using the decimal counting system (0 .. 9). For every **%d** in the format string you must tell **printf()** what variable you want it replaced with. Here's some more use of **printf()** in code:

Example 3-4. more_printf.c

```
int
main()
{
  int one = 1;
  int two = 2;
  int three = 4;  /* the values are unimportant here */

  printf( "one ==\t%d\ntwo ==\t%d\nthree ==\t%d\n", one, two, three );

  return 0;
}
```

3.5. Simple arithmetic

We mentioned at the start of this chapter that point of a program usually involves performing actions on data. By using standard mathematical symbols, arithmetic in C is easily readable.

Example 3-5. wages.c

```
int
main()
{
  int hours_per_day;
  int days_per_week;

  hours_per_day = 8;
  days_per_week = 5;

  printf("I work %d hours a week.\n", (days_per_week * hours_per_day) );
```

```
  printf("%d %d hour days\n", days_per_week, hours_per_day);

  return 0;
}
```

3.6. Global Variables

These have there place but are often used to fix badly written code. If two functions need to operate on a variable you should use pointers to share this variable rather than make it available to *every* function.

3.7. Static Variables

text.

3.8. Constant Variables

A good rule to follow is to never use numbers other than **1** and **0** in your code. If you require another numeric constant you should make it a **const** variable; this way it gets a nice meaningful name. The number 40 has little meaning, however, the identifier **HOURS_WORKED_PER_WEEK** tells us something about what a function is doing. Another benefit is that you can change the value of a const variable in one place rather than having to change all occurrences of 40. Using the latter method it is easy to make a mistake by changing an unrelated occurrence of **40** or forgetting to change an occurrence.

Chapter 4. Flow Control

Taking actions based on decisions

C provides two sytles of decision making: *branching* and *looping*. Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

4.1. Branching

Branching is so called because the program chooses to follow one branch or another. The **if** statement is the most simple of the branching statements. It takes an expression in parenthesis and an statement or block of statements (surrounded by curly braces). *if* the expression is true (evaluates to non-zero) then the statement or block of statements gets executed. Otherwise these statements are skipped. **if** statements take the following form:

```
if (expression)
  statement;
```

or

```
if (expression)
  {
    statement1;
    statement2;
    statement3;
  }
```

Here's a quick code example:

Example 4-1. using_if.c

```
#include <stdio.h>

int
main()
{
  int cows = 6;

  if (cows > 1)
    printf("We have cows\n");

  if (cows > 10)
    printf("loads of them!\n");
```

```
  return 0;
}
```

When compiled and run this program will display:

```
ciaran@pooh:~/book$ gcc -Wall -Werror -o cows using_if.c
ciaran@pooh:~/book$ ./cows
We have cows
ciaran@pooh:~/book$
```

The second **printf()** statement does not get executed because it's expression is false (evaluates to zero).

4.2. if ... else

A second form of **if** statement exists which allows you to also specify a block of code to execute if the test expression is false. This is known as an **if ... else** statement and is formed by placing the reserved word *else* and another block of code after the usual **if** statment. You program will execute one of the two blocks of code based on the test condition after the **if**. Here's what it looks like:

Example 4-2. cows2.c

```
int
main()
{
  int cows = 0;

  if (cows > 1)
    {
      printf("We have cows\n");
      printf("%d cows to be precise\n", cows);
    }
  else
    {
      if (cows == 0)
        printf("We have no cows at all\n");
      else
        printf("We have only one cow\n");
    }

  if (cows > 10)
    printf("Maybe too many cows.\n");

  return 0;
}
```

You should be able to guess the output by now:

```
ciaran@pooh:~/book$ ./cows2
We have no cows at all
ciaran@pooh:~/book$
```

In the last example there was an **if .. else** statement inside another **if .. else** statement. This is perfectly legal in C and is quite common. There is another form of branching you can use but it's a little more complex so we'll leave it to to end of the chapter.

4.3. Loops

Loops provide a way to repeat commands and control how many times they are repeated. Say you wanted to print the alphabet to the screen, you could do this with a call to **printf()**. This is one solution but it doesn't scale very well, what if you wanted to print all the numbers between one and one thousand in a column? this could be handled by one big **printf()** or loads of **printf()** calls but repetitive work should be done by the computer, leaving you more time to work on the interesting parts of your program.

4.4. while

The most basic loop in C is the **while** loop. A **while** statement is like a repeating **if** statement. Like an **If** statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false. If the test condition is false the first time, the statments don't get executed at all. On the other hand if it's test condition never evaluates to false it may continue looping infinitely. To control the number of times a loop executes it's code you usually have at least one variable in the test condition that gets altered in the subsequent block of code. This allows the test condition to become false at some point.

Here's the quick example you are probably expecting. It's a simple guessing game, very simple for the person who is writing the code as they know the answer. When testing this program remember to guess the wrong answer a few times.

Example 4-3. guess_my_number.c

```
#include <stdio.h>


int
main()
{
  const int MAGIC_NUMBER = 6;
  int guessed_number;

  printf("Try to guess what number I'm thinking of\n");
```

```
  printf("HINT: It's a number between 1 and 10\n");

  printf("enter your guess: ");
  scanf("%d", &guessed_number);

  while (guessed_number != MAGIC_NUMBER);
    {
      printf("enter your guess: ");
      scanf("%d", &guessed_number);
    }

  printf("you win.\n")

  return 0;
}
```

The block of code following the **while** statement will be executed repeatedly until the player guesses the number six.

4.5. for

for is similar to **while**, it's just written differently. **for** statements are often used to proccess lists such a range of numbers:

Example 4-4. for_ten.c

```
#include <stdio.h>

int
main()
{
  int i;

  /* display the numbers from 0 to 9 */
  for (i = 0; i < 10; i++)
    printf("%d\n", i);

  return 0;
}
```

4.6. do .. while

do .. while is just like a **while** loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Example 4-5. guess_my_number.c

```
#include <stdio.h>


int
main()
{
  const int MAGIC_NUMBER = 6;
  int guessed_number;

  printf("Try to guess what number I'm thinking of\n");
  printf("HINT: It's a number between 1 and 10\n");

  do
    {
      printf("enter your guess: ");
      scanf("%d", &guessed_number);
    }
  while (guessed_number != MAGIC_NUMBER);

  printf("you win.\n")

  return 0;
}
```

4.7. Switch

The **switch** statement is much like a nested **if .. else** statement. Its mostly a matter of preference which you use, **switch** statement can be slightly more efficient and easier to read.

4.8. The Conditional Operator

The **?:** operator is just like an **if .. else** statement except that because it is an operator you can use it within expressions.

Blah, blah, here's an example:

Example 4-6. apples.c

```
#include <stdio.h>

int
main()
{
  apples = 6;
```

```
  printf("I have %d apple%s\n", apples, (apples == 1) ? "" : "s");

  return 0;
}
```

?: is a *ternary* operator in that it takes three values, this is the only ternary operator C has.

4.9. break & continue

You've see **break** already, we ended each **case** of our **switch** statement with one. **break** exits out of a loop.

continue is similar to break in that it short circuits the execution of a code block but **continue** brings execution back to the start of a loop.

Chapter 5. Pointers

sorting the programmers from the students

5.1. The Basics

A limitation you may have noticed is that functions can only affect your program via their return value, so what do you do when you want a function to alter more than one variable? You use pointers. A pointer is a special kind of variable. Pointers are designed for storing memory address i.e. the address of another variable. Declaring a pointer is the same as declaring a normal variable except you stick an asterisk '*' in front of the variables identifier. There are two new operators you will need to know to work with pointers. The "address of" operator '&' and the "dereferencing" operator '*'. Both are prefix unary operators. When you place an ampersand in front of a variable you will get it's address, this can be store in a pointer. When you place an asterisk in front of a pointer you will get the value at the memory address pointed to. As usual, we'll look at a quick code example to show how simple this is.

Example 5-1. pointers_are_simple.c

```
#include <stdio.h>

int
main()
{
  int my_variable = 6, other_variable = 10;
  int *my_pointer;

  printf("the address of my_variable is    : %p\n", &my_variable);
  printf("the address of other_variable is : %p\n", &other_variable);

  my_pointer = &my_variable;

  printf("\nafter \"my_pointer = &my_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  my_pointer = &other_variable;

  printf("\nafter \"my_pointer = &other_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  return 0;
}
```

The output shows you the address of the two variables, the addresses your system assigns to the variables will be different to mine. In **printf()** you'll notice we used **%p** to display the addresses. This is the conversion specifier for all pointers. Anyway, here's the output I got:

```
the address of my_variable is    : 0xbffffa18
the address of other_variable is : 0xbffffa14

after "my_pointer = &my_variable":
        the value of my_pointer is 0xbffffa18
        the value at that address is 6

after "my_pointer = &other_variable":
        the value of my_pointer is 0xbffffa14
        the value at that address is 10
```

There. That's not too complicated. Once you are comfortable with pointers you're well on your way to mastering C.

5.2. The Address of a Variable

When your program is running and a variable declaration is encountered, you program makes a request for some memory. The operating system finds a spare piece of memory that is large enough and tells your program the address of this piece of memory. Any time your program wants to read the data stored in that variable, it looks at it's memory address and reads the number of bytes equal to the size of the data type of that variable.

If you run the example from the start of this chapter a second time you may or may not get the same result for the addresses, this depends on your system but even if you repeatably get the same addresses right now there is no guarantee that you will get the same result tomorrow, in fact it's rather unlikely.

5.3. Pointers as Function Arguments

One of the best things about pointers is that they allow functions to alter variables outside of there own scope. By passing a pointer to a function you can allow that function to read *and write* to the data stored in that variable. Say you want to write a function that swaps the values of two variables. Without pointers this would be practically impossible, here's how you do it with pointers:

Example 5-2. swap_ints.c

```
#include <stdio.h>

int swap_ints(int *first_number, int *second_number);

int
main()
```

```
{
  int a = 4, b = 7;

  printf("pre-swap values are: a == %d, b == %d\n", a, b)

  swap_ints(&a, &b);

  printf("post-swap values are: a == %d, b == %d\n", a, b)

  return 0;
}

int
swap_ints(int *first_number, int *second_number)
{
  int temp;

  /* temp = "what is pointed to by" first_number; etc... */
  temp = *first_number;
  *first_number = *second_number;
  *second_number = temp;

  return 0;
}
```

As you can see, the function declaration of **swap_ints()** tells GCC to expect two pointers (address of variables). Also, the *address-of* operator (**&**) is used to pass the address of the two variables rather than their values. **swap_ints()** then reads

5.4. Pointer Arithmetic

Arithmetic can be performed on pointers just like any other variable, this is only useful in a few cases though. If you were (for some reason) to divide a pointer by two it would then point to an area of your computers memory that would probably not belong to your program. If your program tried to read or write to this area of memory the text **segmentation fault** will display and your program will abort. A “segmentation fault” occurs when a program tries to access a segment of memory that it does not have permission to access.

There are times however when simple addition can be used on a pointer. We’ll see this in the next chapter when we discuss arrays (multiple variables at consecutive memory addresses). In the case of addition (and subtraction), arithmetic is performed in units equal to the size of the pointers data type.

5.5. Generic Pointers

When a variable is declared as being a pointer to type **void** it is known as a *generic pointer*. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term *Generic pointer*.

This is very useful when you want a pointer to point to data of different types at different times.

Here is some code using a void pointer:

Example 5-3. generic_pointer.c

```
int
main()
{
  int i;
  char c;
  void *the_data;

  i = 6;
  c = 'a';

  the_data = &i;
  printf("the_data points to the integer value %d\n", *(int*) the_data);

  the_data = &c;
  printf("the_data now points to the character %c\n", *(char*) the_data);

  return 0;
}
```

Chapter 6. Structured Data Types

contiguous and structured data

6.1. What is Structured data?

C provides two methods for defining structured, or *aggregate* data types: arrays and structs. Both can contain any of the standard data types including pointers as other structs and arrays. Arrays contain many variables of the same type while structs can contain any mixture of types.

6.2. Arrays

An array is a data type which contains many variables of the same type. Each element of the array is given a number by which you can access that element. For an array of 100 elements, the first element is 0 (zero) and the last is 99. This indexed access makes it very convenient to loop through each element of the array.

6.3. Declaring and Accessing Arrays

Declaring an array is much the same as declaring any other variable except that you must specify the array size. The size (or number of elements) is an integer value placed in square brackets after the arrays identifier.

Example 6-1. first_arrays.c

```
int
main()
{
  int person[10];
  float hourly_wage[4] = {2, 4.9, 10, 123.456};
  int index;

  index = 4;
  person[index] = 56;

  printf("the %dth person is number %d and earns $%f an hour\n",
         (index + 1), person[index], hourly_wage[index]);

  return 0;
}
```

NOTE: it is up to you to make sure you don't try to access an element that is not in the array such as the eleventh element of a ten element array. Attempting to access a value past the end of an array will either crash your program or worse, it could retrieve garbage data without telling you that an error occurred.

6.4. Initialising Arrays

In the above example we initialised the array **hourly_wage** by placing a comma separated list of values in curly braces. Using this method you can initialise as few or as many array elements as you like however you cannot initialise an element without initialising all the previous elements. If you initialise some but not all elements of an array the remaining elements will be automatically initialised to zero.

To get around this inconvenience, a GNU extension to the C language allows you to initialise array elements selectively by number. When initialised by number, the elements can be placed in any order withing the curly braces preceded by **[index]=value**. Like so:

Example 6-2. initialise_array.c

```
#include <stdio.h>

int
main()
{
  int i;
  int first_array[100] = { [90]=4, [0]=5, [98]=6 };
  double second_array[5] = { [3] = 1.01, [4] = 1.02 };

  printf("sure enough, first_array[90] == %d\n\n", first_array[90]);
  printf("sure enough, first_array[99] == %d\n\n", first_array[99]);

  for (i = 0; i < 5; i++}
    printf("value of second_array[%d] is %f\n", i, second_array[i]);

  return 0;
}
```

6.5. Multidimensional Arrays

The array we used in the last example was a *one dimensional* array. Arrays can have more than one dimension, these arrays-of-arrays are called *multidimensional arrays*. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier. A two dimensional array can be though of as a grid of rows and columns.

Example 6-3. number_square.c

```
#include <stdio.h>

const int num_rows = 7;
const int num_columns = 5;

int
main()
{
  int box[num_rows][num_columns];
  int row, column;

  for(row = 0; row < num_rows; row++)
    for(column = 0; column < num_columns; column++)
      box[row][column] = column + (row * num_columns);

  for(row = 0; row < num_rows; row++)
    {
      for(column = 0; column < num_columns; column++)
        {
          printf("%4d", box[row][column]);
        }
      printf("\n");
    }
  return 0;
}
```

If you compile and run this example you'll get a box of numbers like this:

```
   0   1   2   3   4
   5   6   7   8   9
  10  11  12  13  14
  15  16  17  18  19
  20  21  22  23  24
  25  26  27  28  29
  30  31  32  33  34
```

The above array has two dimensions and can be called a doubly subscripted array. GCC allows arrays of up to 29 dimensions although actually using an array of more than three dimensions is very rare.

6.6. Arrays of Characters (Text)

Text in C is represented by a number of consecutive variables of type **char** terminated with the null character **'\0'**.

6.7. Defining data types

The C language provides only the most basic, commonly used types, many languages provide a larger set of types but this is only for convenience. C's way of handling text strings is a good example of this. At times you may think it would be handy if there were other data types which could store complex data. C allows you to define your own.

6.8. Structured Data

In C you can create a new type e.g. "Person". Person can store an int called "age", a string called "name" and another int called "height_in_cm". Here's the code to make this new type:

```
struct Person
{
  char[40] name;
  int age;
  int height_in_cm;
};
```

This code creates a variable called **struct Person**. You can declare variable and pointers to variables of this type in the usual way. Say you declared a variable *john* of type **struct Person**. To access the "age" field you would use **john.age**. I'll make this clearer with a quick example using the previous definition of **struct Person**:

Example 6-4. person_struct.c

```
int
main()
{
  struct Person hero = { 20, "Robin Hood", 191 };
  struct Person sidekick;

  john.age = 31;
  john.name = "John Little"
  john.height_in_cm = 237;

  printf("%s is %d years old and stands %dcm tall in his socks\n",
         sidekick.name, sidekick.age, sidekick.height_in_cm);

  printf( "He is often seen with %s.\n", hero.name );

  return 0;
}
```

When compiled and executed this will display:

```
John Little is 31 years old and stands 237cm tall in his socks
He is often seen with Robin Hood.
```

6.9. Unions

C also supports types that can have dynamic types, a variable that can be and **int** at one point, a **double** later and an **unsigned long long** after that. These data types are declared just like a **struct** except they use the **union** keyword. Their behavior is completely different to a **struct**.

Chapter 7. Run-time Memory Allocation

Requesting memory at run-time

7.1. Why you need this

Often when you write a program you don't actually know how much data is will have to store or process. In previous examples we've read in some text from the user. We've used large character arrays to store this data but what happens if the user enters more text than we can handle? your program crashes. Disaster. At run-time an application can make a request for more memory.

7.2. Dynamic Memory Functions

Glibc provides functions for requesting extra memory, **malloc()** is the fist one we will show you. You must have a pointer to start with.

7.3. Run-time Memory Summary

Forgetting to **free()** memory when your finished with it is one of the worst programming mistakes you can make. Is losing pointers a common problem? pointer falls out of scope?

Chapter 8. Strings and File I/O

Reading and writing to files

8.1. Introduction

Place Holder

Chapter 9. Storage Classes

Changing the behavior of variables

9.1. What are Storage Classes?

You will have noticed that variables in functions lose their values every time the function exists, this is done for reasons of efficiency, the operating system doesn't know if you will need the data again so it releases the memory allocated to your program back to the system.

9.2. auto

By default, variables in C use the **auto** storage class. This is so called because the variables are automatically created when needed and deleted when they fall out of scope.

You can specify a variable to have the **auto** storage class by prefixing the variables declaration with the **auto** keyword but this has no effect, the keyword was introduced into the language for symmetry with the other storage specifiers.

9.3. static

static variables are variables that don't get deleted when they fall out of scope, they are permanent and retain their value between calls to the function. Here's an example:

Example 9-1. list_squares.c

```
#include <stdio.h>

int get_next_square(void);

int
main()
{
  int i;

  for(i = 0; i < 10; i++)
    printf("%6d\n", get_next_square());

  printf("and %6d\n", get_next_square());

  return 0;
}
```

```
int
get_next_square()
{
  static int count = 1;

  count += 1;

  return count * count;
}
```

This will list the squares of the numbers from zero to ten. Zero to nine are printed by the loop and the square of ten is printed afterwards just to show it still has it's value.

9.4. extern

When you declare a variable as **extern** your program doesn't actually reserve any memory for it, **extern** means that the variable already exits *external* to the function or file.

If you want to make a variable available to every file in a project you declare it globally in one file, that is, not inside a function, and add an **extern** declaration of that variable to a header file that is included in all the other files.

9.5. register

The **register** storage class is used as a hint to the compiler that a variable is heavily used and access to it should be optimised if possible. Variables are usually stored in normal memory (RAM) and passed back and forth to the computers processor as needed, the speed the data is sent at is pretty fast but can be improved on. Almost all computer processors contain *cpu registers*, these are memory slots on the actual processor, storing data there gets rid of the overhead of retrieving the data from normal memory. This memory is quite small compared to normal memory though so only a few variables can be stored there. GCC will always make use of registers by deciding what variables it thinks will be accessed often, this works well but will never be perfect because GCC doesn't know the purpose of your program. By using the **register** keyword you can tell GCC what needs to be optimised.

One problem with placing a variable into a cpu register is that you can't get a pointer to it, pointers can only point to normal memory. Because of this restriction GCC will ignore the **register** keyword on variables whos address is taken at any point in the program.

The resulting program will contain a request, on creation of the variable that it be placed in a cpu register, the operating system may ignore or honour this request.

9.6. the restrict type qualifier

This is something to do with pointers, I think it tells the compiler that a specific pointer is the only pointer to a section of memory, the compiler can optimise code better with this knowledge. I think.

9.7. typedef

typedef isn't much like the others, it's used to give a variable type a new name. There are two main reasons for doing this. The most common is to give a name to a struct you have defined so that you can use your new data type without having to always precede it with the struct keyword.

The second use for typedef is for compatibility. Say you want to store a 32-bit number. If you use **int** you are not guaranteed that it will be 32-bit on every machine. To get around this you can use preprocessor directives to selectively typedef a new type to the right size.

Example 9-2. battleships.c

```
#include <stdio.h>

/* type, position coordinates and armament */
struct _ship
{
  int type;
  int x;
  int y;
  int missiles;
};

typedef struct _ship ship;

int
main()
{
  ship battle_ship_1;
  ship battle_ship_2 = {1, 60, 66, 8};

  battle_ship_1.type = 63;
  battle_ship_1.x = 54;
  battle_ship_1.y = 98;
  battle_ship_1.missiles = 12;

  /* More code to actually use this data would go here */

  return 0;
}
```

Chapter 10. The C Preprocessor

When & how to use them

10.1. What is the C Prepressor

The C Preprocessor is a simple macro-expander that is run on source code files before passing them to the compiler. Lines that begin with the hash symbol **'#'** are directives to the C preprocessor.

When you create a macro you assign a name to a C expression. You can then use this name in your code just as you would have used the expression. The preprocessor replaces all occurences of that name with the expression.

10.2. What is it used for?

Macros are snippets of code that get processed before compilation. This is done by the *C preprocessor*, **#define** statements are macros. Take a look at this piece of code:

Example 10-1. box_of_stars.c

```
#define SIZE_OF_SQUARE 4

int
main()
{
  int i, j;

  for(i = 0; i < SIZE_OF_SQUARE; i++)
  {
    for(j = 0; j < SIZE_OF_SQUARE; j++)
    {
      printf("*"); // print an asterisk for each column
    }

    printf("\n"); // and a newline at the end of each row
  }
}
```

The output of this code will be a box:

```
****
****
****
****
```

The C preprocessor simply replaces the macro **SIZE_OF_BOX** with the value “4”. This very useful for two reasons:

- firstly the size of the box can be changed by just editing one line. This isn’t a huge advantage in the above example as there are just two uses of **SIZE_OF_BOX** but in larger programs this make life much easier and removes the possibility of forgetting to change one of the values.
- Secondly it makes the code more readable, meaningful names can be given to values such as **#define PI 3.142857143**.

10.3. Some sample macros

Some of the small function in glibc are implemented as macros, **getc()** is one

10.4. Caveats for macros

Macros *can* be miss-used and it’s hard to catch the bugs because the macro no longer exists when the code gets to the compiler. The most error is the macro argument with side effect. Take the this small example:

Example 10-2. max_macro.c

```
#define MAX(a, b) (a > b ? a : b)

int
main()
{
  int cows = 10, sheep = 12;

  printf("we have %d of our most common animal\n", MAX(cows, sheep));

  return 0;
}
```

We compile and execute this code and get:

```
ciaran@pooh:~/book$ ./a.out
we have 12 of our most common animal
ciaran@pooh:~/book$
```

Yup, everything looks good. Try this next example:

Example 10-3. max_macro_problem.c

```
#define MAX(a, b) (a > b ? a : b)

int
main()
{
  int cows = 10, sheep = 12;

  printf("We have %d of our most common animal\n", MAX(cows, sheep));

  printf("Hang on, we just bought another one.\n");
  printf("Now we have %d.\n", MAX(cows, ++sheep));

  return 0;
}
```

Can you see what's going to happen?

```
ciaran@pooh:~/book$ ./a.out
We have 12 of our most common animal
Hang on, we just bought another one.
Now we have 14.
ciaran@pooh:~/book$
```

When the text substitution occurs there will be two instances of **++sheep**. Another more sinister way for this bug may manifest itself is when you pass use a function as an argument to a macro. If the function modifies a global or static variable then this modification may occur multiple times. These bugs can be *very* hard to find, the code is perfectly valid so the compiler has nothing to complain about, the bug will only be noticed at run time and wont occur every time the macro is called, only when it is called with an argument that has a side effect.

10.5. Are macros necessary?

The preprocessor was commonly used to make up for small deficiencies of the language, however, as the language has evolved these defiances have be all but done away. It's still good to know how to use and understand preprocessor macros, they are very common. Macros have been part of the language for longer than their replacements and people have gotten used to them.

10.6. Replacing Simple Macros

If you are thinking that a **const int** global variable could replace a simple **#define** you are right. **const** variables have some advantages, one small advantage is that you can get their address when you need to pass around a pointer to their value, in this way they are more flexible than macros. If you don't take the address of the **const** variable then GCC can optimise it to a level similar to a **#define**.

10.7. Replacing Complex Macros

Complex macros, that is functions implemented as macros, can be replaced by **inline** functions.

Chapter 11. Variable Length Arguments

The VA_ARGS macros etc.

11.1. What are Variable Length Arguments?

Variable length argument lists are something you have already come across, think of **printf()**, how would you write the prototype for this function when you don’t know how many arguments will be passed to it.

Chapter 12. Tricks with Functions

pointers to functions

12.1. What are Virtual Functions?

Another use of the **void** data type is for making pointers to functions. This is a fairly advanced programming technique but a very useful one once you become comfortable with it.

Here is an example of a function pointer:

Example 12-1. virtual_function.c

```
int
main()
{
  /* oh, crap, better go write one... */

  return 0;
}
```

12.2. Nesting functions

GCC permits functions to be defined within other functions. Functions defined like this are known as nested functions and obey the same scoping rules as variables. When the parent function exits, the child function falls out of scope and is unavailable.

12.3. The Benefits of Nested Functions

Probably the main reason for nested functions being allowed by GCC for flexibilities sake although small performance increases can be gained by using them correctly. Nested functions obey *Lexical scoping*, they have access to the variables of the function that contains them.

For this reason, they can accomplish tasks that would usually require functions taking pointers as arguments. There is a slight performance loss when pointers are used because a variable that has a pointer cannot be stored in a machine register. Pointers never point to machine registers so how would the pointer work?

12.4. Declaring and Defining Nested Functions

If you are defining a function at the

Example 12-2. simple nested function

```
#include <stdio.h>

int
main()
{
  int swap (int *a, int *b)
    {
      int c;

      c = *a;
      *a = *b;
      *b = c;

      return 0;
    }

  int first = 12, second = 34;

  printf("f is %d and s is %d\n", first, second);

  swap(&first, &second);

  printf("f is %d and s is %d\n", first, second);

  return 0;
}
```

You don’t have to declare nested functions like you do normal functions however you can if you like. The only reason for doing so would be for the sake of readability, you might like the function definition to appear near where it is used. It’s up to you, but if you do decide to declare you nested function you must explicitly declare it as **auto**.

12.5. Scope

Nested functions have local scope, declaring a nested function as **extern** is will cause an error. **static** and **inline** are both valid however the meaning of **static** escapes me.

Chapter 13. Taking Command Line Arguments

13.1. How does C handle command line arguments?

A program starts by the operating system *calling* a programs **main()** function. Every one of your programs so far have defined **main()** as a function taking no arguments but this is not always the case. **main()** is the only function in C that can be defined in multiple ways. It can take no arguments, two arguments or three arguments. The two and three argument forms allow it to receive arguments from the shell. The three argument form is not particularly useful and is never necessary, we'll cover it briefly at the end of this chapter.

The two argument form takes an **int** and an array of strings. When defining **main()** you can give these arguments any name but it is convention to call them **argc** and **argv[]**. The first argument holds a count of how many elements there are in the array of strings passed as the second argument. The array is always null terminated so **argv[argc] == NULL**.

Here's a short program demonstrating the use of

Example 13-1. list_args.c

```
int
main(int argc, char *argv[])
{
  int i;

  for(i = 0; i < argc; i++)
    printf("argv[%d] == %s\n", i, argv[i]);

  return 0;
}
```

to be passed to **main()** via two arguments: an **int** and a ***char[]** (an array of strings). The **int** is usually called **argc** which is short for "argument count", as the name suggests, it stores the number of arguments passed to **main()**. The second argument, usually called **argv** is an array of strings.

13.2. Argp

C's method of getting command line arguments is pretty simple but when your program has a lot of options it can get complex. To solve this, Glibc provides a series of functions to perform command tasks

for you. The “argp_*” functions perform much of the work for you and they do it in a standard way which makes you program more familiar to users. Here’s an short program using argp:

Example 13-2. simple_argp.c

```
/* put a tiny argp program here */
```

When you run this program you will see...

13.3. Using More of the Argp Functionality

Here’s a longer program, it uses four global variables to store information about your program:

Example 13-3. better_argp.c

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

const char *argp_program_version = "simple_argp 0.1";
const char *argp_program_bug_address =
          "<some_email_address@you_care_about.com>";

static char doc[] =
"short program to show the use of argp\nThis program does little";

static char args_doc[] = "ARG1 ARG2";

/* initialise an argp_option struct with the options we except */
static struct argp_option options[] =
{
  {"verbose", 'v', 0,      0, "Produce verbose output" },
  {"output",  'o', "FILE", 0, "Output to FILE" },
  { 0 }
};

/* Used by 'main' to communicate with 'parse_opt'. */
struct arguments
{
  char *args[2];                /* ARG1 & ARG2 */
  int silent, verbose;
  char *output_file;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  /* Get the INPUT argument from 'argp_parse', which we
     know is a pointer to our arguments structure. */
```

```
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'q': case 's':
      arguments->silent = 1;
      break;
    case 'v':
      arguments->verbose = 1;
      break;
    case 'o':
      arguments->output_file = arg;
      break;

    case ARGP_KEY_ARG:
      if (state->arg_num >= 2)
        /* Too many arguments. */
        argp_usage (state);

      arguments->args[state->arg_num] = arg;

      break;

    case ARGP_KEY_END:
      if (state->arg_num < 2)
        /* Not enough arguments. */
        argp_usage (state);
      break;

    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  struct arguments arguments;

  /* Default values. */
  arguments.silent = 0;
  arguments.verbose = 0;
  arguments.output_file = "-";

  /* Parse our arguments; every option seen by 'parse_opt' will
     be reflected in 'arguments'. */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  printf ("ARG1 = %s\nARG2 = %s\nOUTPUT_FILE = %s\n"
          "VERBOSE = %s\nSILENT = %s\n",
```

```
          arguments.args[0], arguments.args[1],
          arguments.output_file,
          arguments.verbose ? "yes" : "no",
          arguments.silent ? "yes" : "no");

  exit (0);
}
```

This is pretty simple. no?

13.4. Environment Variables

BLAH, talk about how to use the three argument form and the other way of getting at environment variables, show a toy example.

Chapter 14. Using and Writing Libraries

Reusable compiled code

14.1. What are Libraries?

Libraries are like programs except they don't contain a **main()** function for execution to begin at. The functions in libraries can be used in other applications by linking the application to the library.

14.2. Using Libraries

To link with a library you must use the **-l} s**itch followed by the library name. Glibc's math library is called libm.so, you don't need the **lib}** r **.so}** arts to to use this library you add the switch **-lm}** o your compilation line.

14.3. Stages of Compilation

Compiling a program involves many tools, GCC takes care of this by calling other programs to handle each stage of the process. The three main stages are *preprocessing*, *compilation* and *linking*. In C code, lines that begin with the *hash* symbol "#" are commands for the preprocessor, GCC includes a preprocessor called CPP (C Preprocessor). **#define** and **#include** are by far the most common preprocessor commands The compilation process is broken down into many smaller stages. One of these stages is confusingly called *compilation*. Compilation is the process of converting source code to *object code*. If you invoke GCC with "-c" it will When programs become large it can take time to compile them, by splitting a program into smaller files you can re-compile only the files that you have changed. First you must tell gcc to only compile the source files. This

14.4. Writing a library

Writing a library is similar to writing a program. The first obvious difference is that there is no **main()**. Libraries are very handy for functions you use regularly or functions you think others may find useful.

14.5. Dynamic or Static

Libraries can be linked in a *dynamic* or *static* way. A staticly linked library gets compiled into your application. Dynamic linking is a newer method that allows an application to link with a library at run-time, this has many advantages. For a start, the library can be updated without having to recompile the application and vice versa.

Chapter 15. Writing Good Code

Considerations for important projects

15.1. Readability

This is one of the most important qualities of good code, luckily, it is an art that becomes natural with practice. One of the biggest detractors from readability is “clever” code. Scrunching multiple operations into one line or statement may have seemed like an achievement but when someone else tries to read it it will slow them down if they have to unravel these operations in there head.

Chapter 16. Speed

Easy optimisations: Low hanging fruit

C is well know as the fastest high-level language available

16.1. About Optimising

There are two times you will optimise your code: while you're writing it and after it's performance disappoints you.

As a rule, it is said that ninety percent of your an applications running time is taken up by ten percent of it's code. There is little point in optimising a function that is rarely called.

16.2. What are function attributes?

Function attributes are a GNU extension to the C language. They allow you to give GCC more information about a function. This information can be used for many purposes including optimisation and stricter checking.

16.3. Function Attribute Syntax

Function attributes are specified as part of a function declaration. After the closing parenthesis of the functions arguments the keyword **__attribute__** followed by the desired attributes in a set of double parenthesis. Here's a function with the *pure* attribute.

```
int my_func(int first, int second) __attribute__ ((pure));
```

Functions can have multiple attributes, to do this, separate the attributes with commas inside the double parenthesis.

16.4. What are pure and const?

A **pure** function is one which do not affect anything outside of it's own scope. This means it may read global variables or variables to which it was passed a pointer but it may not write to such variables. It should not read from **volatile** variables or external resources (such as files).

const is a stricter version of **pure**, it tells GCC that a function will not read any data other that of variables that are passed to it. Data cannot be read by dereferencing a pointer passed to a **const** function.

The only effect a **pure** or **const** function can have on your program is it's return value. Having such a function return **void** would make it pointless.

GCC can use this information to perform *common subexpression elimination* (!). This means it may call the function fewer times than it was told to as it knows the outcome will be the same each time. For example: if you had a function which converted Celsius to Fahrenheit, and it was placed in a loop calculating the same value each time, GCC would could replace this function call with the value returned. GCC knows this is safe if the conversion function is **const**.

Appendix A. GNU Free Documentation License

Version 1.2, November 2002

Free Software Foundation, Inc.
59 Temple Place, Suite 330,
Boston,
MA
02111-1307
USA

A.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a

world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which

do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to

download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”,

and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

A.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

A.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM (How to use this License for your documents)

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME.
>
> 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, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace The “with...Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.