Edition 4.1

The GNU C Programming Tutorial

Mark Burgess
Faculty of Engineering, Oslo College

Ron Hale-Evans

Preface

This book is a tutorial for the computer programming language C. Unlike BASIC or Pascal, C was not written as a teaching aid, but as a professional tool. Programmers love C! Moreover, C is a standard, widely-used language, and a single C program can often be made to run on many different kinds of computer. As Richard M. Stallman remarks in *GNU Coding Standards*, “Using another language is like using a non-standard feature: it will cause trouble for users.” (See `http://www.gnu.org/prep/standards_toc.html`.)

Skeptics have said that everything that can go wrong in C, does. True, it can be unforgiving, and holds some difficulties that are not obvious at first, but that is because it does not withhold its powerful capabilities from the beginner. If you have come to C seeking a powerful language for writing everyday computer programs, you will not be disappointed.

To get the most from this book, you should have some basic computer literacy — you should be able to run a program, edit a text file, and so on. You should also have access to a computer running a GNU system such as GNU/Linux. (For more information on GNU and the philosophy of free software, see `http://www.gnu.org/philosophy/`.)

The tutorial introduces basic ideas in a logical order and progresses steadily. You do not need to follow the order of the chapters rigorously, but if you are a beginner to C, it is recommended that you do. Later, you can return to this book and copy C code from it; the many examples range from tiny programs that illustrate the use of one simple feature, to complete applications that fill several pages. Along the way, there are also brief discussions of the philosophy behind C.

Computer languages have been around so long that some jargon has developed. You should not ignore this jargon entirely, because it is the language that programmers speak. Jargon is explained wherever necessary, but kept to a minimum. There is also a glossary at the back of the book.

The authors of this book hope you will learn everything you need to write simple C programs from this book. Further, it is released under the GNU Free Documentation License, so as the computers and robots in the fantasies of Douglas Adams say, “Share and Enjoy!”

The first edition of this book was written in 1987, then updated and rewritten in 1999. It was originally published by Dabs Press. After it went out of print, David Atherton of Dabs and the original author, Mark Burgess, agreed to release the manuscript. At the request of the Free Software Foundation, the book was further revised by Ron Hale-Evans in 2001 and 2002.

The current edition is written in Texinfo, which is a documentation system using a single source file to produce both online information and printed output. You can read this tutorial online with either the Emacs Info reader, the stand-alone Info reader, or a World Wide Web browser, or you can read it as a printed book.

1 Introduction

What is a high-level language? Why is C unusual?

Any sufficiently complex object has levels of detail; the amount of detail we see depends on how closely we scrutinize the object. A computer has many levels of detail.

The terms *low level* and *high level* are often used to describe these layers of complexity in computers. The low level is buried in the computer's microchips and microcircuits. The low level is the level at which the computer seems most primitive and mechanical, whereas the high level describes the computer in less detail, and makes it easier to use.

You can see high levels and low levels in the workings of a car. In a car, the nuts, bolts, and pistons of the low level can be grouped together conceptually to form the higher-level engine. Without knowing anything about the nuts and bolts, you can treat the engine as a *black box*: a simple unit that behaves in predictable ways. At an even higher level (the one most people use when driving), you can see a car as a group of these black boxes, including the engine, the steering, the brakes, and so on. At a high level, a computer also becomes a group of black boxes.

C is a high-level language. The aim of any high-level computer language is to provide an easy, natural way to give a list of instructions (a computer program) to a computer. The native language of the computer is a stream of numbers called *machine language*. As you might expect, the action resulting from a single machine language instruction is very primitive, and many thousands of them can be required to do something substantial. A high-level language provides a set of instructions you can recombine creatively and give to the imaginary black boxes of the computer. The high-level language software will then translate these high-level instructions into low-level machine language instructions.

1.1 The advantages of C

C is one of a large number of high-level languages designed for *general-purpose programming*, in other words, for writing anything from small programs for personal amusement to complex industrial applications.

C has many advantages:

- Before C, machine-language programmers criticized high-level languages because, with their black box approach, they shielded the user from the working details of the computer and all its facilities. C, however, was designed to give access to any level of the computer down to raw machine language, and because of this, it is perhaps the most flexible high-level language.
- C has features that allow the programmer to organize programs in a clear, easy, logical way. For example, C allows meaningful names for variables without any loss of efficiency, yet it gives a complete freedom of programming style, including flexible ways of making decisions, and a set of flexible commands for performing tasks repetitively (`for`, `while`, `do`).
- C is succinct. It permits the creation of tidy, compact programs. This feature can be a mixed blessing, however, and the C programmer must balance simplicity and readability.

- C allows commands that are invalid in other languages. This is no defect, but a powerful freedom which, when used with caution, makes many things possible. It does mean that there are concealed difficulties in C, but if you write carefully and thoughtfully, you can create fast, efficient programs.
- With C, you can use every resource your computer offers. C tries to link closely with the local environment, providing facilities for gaining access to common peripherals like disk drives and printers. When new peripherals are invented, the GNU community quickly provides the ability to program them in C as well. In fact, most of the GNU project is written in C (as are many other operating systems).

For the reasons outlined above, C is the preeminent high-level language. Clearly, no language can guarantee good programs, but C can provide a framework in which it is easy to program well.

1.2 Questions for Chapter 1

1. Explain the distinction between high levels and low levels.
2. What is a "black box"?
3. Name a few advantages to programming in the C language.

2 Using a compiler

How to use a compiler. What can go wrong.

The *operating system* is the layer of software that drives the hardware of a computer and provides the user with a comfortable work environment. Operating systems vary, but most have a *shell*, or text interface. You use the GNU shell every time you type in a command that launches an email program or text editor under GNU.

In the following sections of this chapter, we will explore how to create a C program from the GNU shell, and what might go wrong when you do.

2.1 Basic ideas about C

First a note about a programming language that is different from the C programming language, the GNU shell. When you enter commands in the GNU shell, they are executed immediately. Moreover, the shell is a programming language, in that the commands you type are a program, because you can also create a text file containing many shell commands. When you run this file, the commands will be executed in sequence.

On the other hand, consider C. While a shell command file can be executed directly, a C program must be created in two stages:

1. First, the program is written in the form of text files with a text editor such as GNU Emacs. This form of the program is called the *source code*. A computer cannot execute source code directly.
2. Second, the completed source code is processed with a *compiler* — a program that generates a new file containing a machine-language translation of the source code. This file is called an *executable file*, or *executable*. The executable file is said to have been *compiled* from the source code.

To run the compiled program, you must usually type the name of the executable file preceded by a period and a slash, as in this example:

```
./myprogram
```

The "dot-slash" prefix tells the GNU shell to look in the current directory for the executable. You usually do not need to type '`./`' in front of commands for programs that came with your GNU system, such as `emacs`, because the computer already knows where to look for the executables of those programs, which were placed in special directories when your GNU system was installed.

A C program is made up of, among other components, variables and functions. A *variable* is a way to hold some data which may vary, hence the name. For example, a variable might hold the number 17, and later the number 41. Another variable might hold the word "Sue".

A *function* is a segment of text in the source code of a program that tells the computer what to do. Programming consists, in large part, of writing functions.

2.2 The compiler

When you compile a program, the compiler usually operates in an orderly sequence of phases called *passes*. The sequence happens approximately like this:

1. First, the compiler reads the source code, perhaps generating an intermediate code (such as *pseudo-code*) that simplifies the source code for subsequent processing.
2. Next, the compiler converts the intermediate code (if there is any) or the original source code into an *object code* file, which contains machine language but is not yet executable. The compiler builds a separate object file for each source file. These are only temporary and are deleted by the compiler after compilation.
3. Finally, the compiler runs a *linker*. The linker merges the newly-created object code with some standard, built-in object code to produce an executable file that can stand alone.

GNU environments use a simple command to invoke the C compiler: `gcc`, which stands for "GNU Compiler Collection". (It used to stand for "GNU C Compiler", but now GCC can compile many more languages than just C.) Thus, to compile a small program, you will usually type something like the following command:

    gcc file_name

On GNU systems, this results in the creation of an executable program with the default name '`a.out`'. To tell the compiler you would like the executable program to be called something else, use the '`-o`' option for setting the name of the object code:

    gcc -o program_name file_name

For example, to create a program called '`myprog`' from a file called '`myprog.c`', write

```
gcc -o myprog myprog.c
```

To launch the resulting program '`myprog`' from the same directory, type

```
./myprog
```

2.3 File names

GCC uses the following file name conventions:

Source code file	*program_name*`.c`
Object file	*program_name*`.o`
Executable file	*program_name* (no ending)
Header file	*name*`.h`
Library file	`lib`*name*`.a` or `lib`*name*`.so`

The file name endings, or *file extensions*, identify the contents of files to the compiler. For example, the '`.c`' suffix tells the compiler that the file contains C source code, and the other letters indicate other kinds of files in a similar way.

2.4 Errors

Errors are mistakes that programmers make in their code. There are two main kinds of errors.

- *Compile-time errors* are errors caught by the compiler. They can be *syntax errors*, such as typing `fro` instead of `for`, or they can be errors caused by the incorrect construction of your program. For example, you might tell the compiler that a certain variable is an integer, then attempt to give it a non-integer value such as 5.23. (See Section 2.4.2 [Type errors], page 6.)

  The compiler lists all compile-time errors at once, with the line number at which each error occurred in the source code, and a message that explains what went wrong.

  For example, suppose that, in your file '`eg.c`' you write

  ```
  y = sin (x];
  ```

  instead of

  ```
  y = sin (x);
  ```

  (By the way, this is an example of *assignment*. With the equals sign ('`=`'), you are *assigning* the variable `y` (causing the variable `y` to contain) the sine of the variable `x`. This is somewhat different from the way equals signs work in mathematics. In math, an equals sign indicates that the numbers and variables on either side of it are *already* equal; in C, an equals sign *makes* things equal. Sometimes it is useful to think of the equals sign as an abbreviation for the phrase "becomes the value of".)

  Ignore the syntactic details of the statements above for now, except to note that closing the `(x)` with a square bracket instead of a parenthesis is an error in C. Upon compilation, you will see something like this error message:

  error

  ```
  eg.c: In function 'main':
  eg.c:8: parse error before ']'
  ```

  (If you compile the program within Emacs, you can jump directly to the error. We will discuss this feature later. See Chapter 23 [Debugging], page 223, for more information.)

  A program with compile-time errors will cause the compiler to halt, and will not produce an executable. However, the compiler will check the syntax up to the last line of your source code before stopping, and it is common for a single real error, even something as simple as a missing parenthesis, to result in a huge and confusing list of nonexistent "errors" from the compiler. This can be shocking and disheartening to novices, but you'll get used to it with experience. (We will provide an example later in the book. See Chapter 23 [Debugging], page 223.)

  As a rule, the best way to approach this kind of problem is to look for the *first* error, fix that, and then recompile. You will soon come to recognize when subsequent error messages are due to independent problems and when they are due to a cascade.

- *Run-time errors* are errors that occur in a compiled and running program, sometimes long after it has been compiled.

  One kind of run-time error happens when you write a running program that does not do what you intend. For example, you intend to send a letter to all drivers whose licenses will expire in June, but instead, you send a letter to all drivers whose licenses will *ever* expire.

Another kind of run-time error can cause your program to *crash*, or quit abruptly. For example, you may tell the computer to examine a part of its memory that doesn't exist, or to divide some variable by zero. Fortunately, the GNU environment is extremely stable, and very little will occur other than an error message in your terminal window when you crash a program you are writing under GNU.

If the compilation of a program is successful, then a new executable file is created.

When a programmer wishes to make alterations and corrections to a C program, these must be made in the source code, using a text editor; after making the changes, the programmer must recompile the program, or its salient parts.

2.4.1 Typographical errors

The compiler can sometimes fail for very simple reasons, such as typographical errors, including the misuse of upper- and lower-case characters. The C language is *case-sensitive*. Unlike languages such as Pascal and some versions of BASIC, C distinguishes between upper- and lower-case letters, such as '`A`' and '`a`'. If a letter is typed in the wrong case in a critical place in the source code, compilation will fail. This is a potential source of errors that are difficult to find.

2.4.2 Type errors

C supports a variety of *variable types* (different kinds of variables for different kinds of data), such as `integer` for integer numbers, and `float` for numbers with fractional parts. You can even define your own types, such as `total` for a sum, or `surname` for someone's last name. You can also convert a variable of one type into other types. (This is called *type coercion*.) Consequently, the type of a variable is of great importance to the compiler.

C requires us to list the names and types of all variables that will be used in a program, and provide information about where they are going to be used. This is called *declaring* variables. If you fail to declare a variable, or use it as if it were a different type from the type it is declared to be, for example, by assigning a non-integer value to an integer variable, you will receive a compile-time error.

See Chapter 5 [Variables and declarations], page 19, for more information on variable declarations. See Chapter 3 [The form of a C program], page 9, for some simple examples of variable declarations.

2.5 Questions for Chapter 2

1. What is a compiler?
2. How does one run a C program?
3. How does one usually compile a C program?
4. Are upper and lower case equivalent in C?
5. What the two main kinds of error that can occur in a program?

6. If you had some C source code that you wished to call "accounts", under what name would you save it?
7. What would be the name of the executable file for the program in the last question?
8. How would you run this program?

3 The form of a C program

What goes into a C program? What does one look like?

The basic building block of a C program is the *function*. Every C program is a collection of one or more functions. Functions are made of variable declarations and *statements*, or complex commands, and are surrounded by curly brackets ('{' and '}').

One and only one of the functions in a program must have the name `main`. This function is always the starting point of a C program, so the simplest C program is a single function definition:

```
main ()
{
}
```

The parentheses '`()`' that follow the name of the function must be included. This is how C distinguishes functions from ordinary variables.

The function `main` does not need to be at the top of a program, so a C program does not necessarily start at line 1, but wherever the function called `main` is located. The function `main` cannot be *called*, or started, by any other function in the program. Only the operating system can call `main`; this is how a C program is started.

The next most simple C program is perhaps a program that starts, calls a function that does nothing, and then ends.

```
/******************************************************/
/*                                                    */
/* Program : do nothing                               */
/*                                                    */
/******************************************************/

main()                          /* Main program */
{
  do_nothing();
}

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

do_nothing()                 /* Function called */
{
}
```

(Any text sandwiched between '`/*`' and '`*/`' in C code is a comment for other humans to read. See the section on comments below for more information.)

There are several things to notice about this program.

First, this program consists of two functions, one of which calls the other.

Second, the function `do_nothing` is called by simply typing the main part of its name followed by '`()`' parentheses and a semicolon.

Third, the semicolon is vital; every simple statement in C ends with one. This is a signal to the compiler that the end of a statement has been reached and that anything that follows is part of another statement. This signal helps the compiler diagnose errors.

Fourth, the curly bracket characters '{' and '}' outline a *block* of statements. When this program meets the closing '}' of the second function's block, it transfers control back to 'main', where it meets another '}', and the program ends.

3.1 A word about style

The code examples above are simple, but they illustrate the *control flow* of a C program, or the order in which its statements are executed. You should note that these programs are written in "old-fashioned" C, as the language existed before ANSI Standard C — the version in which most C programs are now written. The above programs are also missing several key elements that most C programs have, such as header files and function prototypes. Finally, they do not show good style; if you wish to submit programs you write to the Free Software Foundation, you should consult its advice on how best to use the C language.

You may wonder why we chose old-style C for these first few examples, even though people proverbially learn best what they learn first. We did so because pre-ANSI C is considerably simpler than the present form, and also because as you develop as a C programmer, you will probably run across some old C code that you will want to read.

You may also wonder why a savvy programmer would want to follow the ANSI Standard, which was drafted by committee, or even the GNU guidelines. Isn't programming free software all about freedom? Yes, but following the ANSI Standard ensures that your code can be easily compiled on many other computer platforms, and the GNU guidelines ensure that your code can be read by other programmers. (We will introduce good C style in our examples soon. Meanwhile, you can examine the GNU guidelines later in the book. See Chapter 22 [Style], page 219.)

3.2 Comments

Annotating programs.

Comments are a way of inserting remarks and reminders into code without affecting its behavior. Since comments are only read by other humans, you can put anything you wish to in a comment, but it is better to be informative than humorous.

The compiler ignores comments, treating them as though they were *whitespace* (blank characters, such as spaces, tabs, or carriage returns), and they are consequently ignored. During compilation, comments are simply stripped out of the code, so programs can contain any number of comments without losing speed.

Because a comment is treated as whitespace, it can be placed anywhere whitespace is valid, even in the middle of a statement. (Such a practice can make your code difficult to read, however.)

Any text sandwiched between '/*' and '*/' in C code is a comment. Here is an example of a C comment:

```
/* ...... comment ......*/
```

Comments do not necessarily terminate at the end of a line, only with the characters '`*/`'. If you forget to close a comment with the characters '`*/`', the compiler will display an '`unterminated comment`' error when you try to compile your code.

3.3 Example 1

```
#include <stdio.h>      /* header file */

main ()   /* Trivial program */

{

/* This little line has no effect */
/* This little line has none */
/* This little line went all the way down
   to the next line,
   And so on...
   And so on...
   And so on... */

  do_little();

  printf ("Function 'main' completing.\n");
}

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

/* A bar like the one above can be used to */
/* separate functions visibly in a program */


do_little ()
{

/* This function does little. */

  printf ("Function 'do_little' completing.\n");
}
```

Again, this example is old-fashioned C, and in mediocre style. To make it more compliant with the ANSI Standard and GNU guidelines, we would declare the variable type each function returns (`int` for `main`, which also requires an `exit` or `return` statement), and we would create function prototypes at the beginning of the file. (See Chapter 4 [Functions], page 13.)

3.4 Questions for Chapter 3

1. What is a block?
2. Does a C program start at the beginning? Where is the beginning?

3. What happens when a program comes to a '`}`' character? What does this character signify?
4. What vital piece of punctuation goes at the end of every simple C statement?
5. What happens if a comment is not ended? That is if the programmer types '`/*`' .. to start but forgets the ..'`*/`' to close?

4 Functions

Solving problems and getting results.

A function is a section of program code that performs a particular task. Making functions is a way of isolating one section of code from other independent sections. Functions allow a programmer to separate code by its purpose, and make a section of code *reusable* — that is, make it so the section can be called in many different contexts.

Functions should be written in the following form:

```
type function_name (type parameter1_name, type parameter2_name, ...)

{
  variable declarations

  statements
  ...
  ...
  ...
}
```

You may notice when reading the examples in this chapter that this format is somewhat different from the one we have used so far. This format conforms to the ANSI Standard and is better C. The other way is old-fashioned C, although GCC will still compile it. Nevertheless, GCC is not guaranteed to do so in the future, and we will use ANSI Standard C in this text from now on.

As shown above, a function can have a number of *parameters*, or pieces of information from outside, and the function's *body* consists of a number of declarations and statements, enclosed by curly brackets: '`{...}`'.

4.1 Function names

Every function has a name by which it is known to the rest of the program. The name of a function in C can be anything from a single letter to a long word. The ANSI Standard, however, only guarantees that C will be able to distinguish the first 31 letters of *identifiers*, or function and variable names. (Identifiers are therefore said to have 31 *significant characters*.) In some cases, identifiers may have as few as six significant characters, to stay compatible with older linkers, but this part of the ANSI Standard is becoming obsolete.

A function name must begin with an alphabetic letter or the underscore '`_`' character, but the other characters in the name can be chosen from the following groups:

- Any lower-case letter from '`a`' to '`z`'
- Any upper-case letter from '`A`' to '`Z`'
- Any digit from '`0`' to '`9`'
- The underscore character '`_`'

Note that with GCC, you can also use dollar signs ('`$`') in identifiers. This is one of GCC's extensions to the C language, and is not part of the ANSI standard. It also may not be supported under GCC on certain hardware platforms.

4.2 Function examples

Here is an example of a function that adds two integers and prints the sum with C's "print formatted" function named `printf`, using the characters '`%d`' to specify integer output.

```
void add_two_numbers (int a, int b)               /* Add a and b */
{
  int c;

  c = a + b;
  printf ("%d\n", c);
}
```

The variables `a` and `b` are parameters passed in from outside the function. The code defines `a`, `b`, and `c` to be of type `int`, or integer.

The function above is not much use standing alone. Here is a `main` function that calls the `add_two_numbers` function:

```
int main()
{
  int var1, var2;

  var1 = 1;
  var2 = 53;

  add_two_numbers (var1, var2);
  add_two_numbers (1, 2);

  exit(0);
}
```

When these functions are incorporated into a C program, together they print the number 54, then they print the number 3, and then they stop.

4.3 Functions with values

In mathematics, a function takes one or more values and calculates, or *returns*, another value. In C, some functions return values and others do not; whether a function you write does or does not will depend on what you want the function to do. For example, a function that calculates a value should probably return that value, while a function that merely prints something out may not need to.

The `add_two_numbers` function above did not return a value. We will now examine a function that does.

Here is an example of calling a function that returns a value:

```
bill = calculate_bill (data1, data2, data3);
```

When this statement is executed, control is passed to the function `calculate_bill`, that function executes, and then it returns control and some value to the original statement. The value returned is assigned to `bill`, and the program continues.

In C, returning a value from a function is a simple matter. Consider the function `calculate_bill` as it might be written in a program that contains the statement above:

```
int calculate_bill (int a, int b, int c)
{
  int total;

  total = a + b + c;
  return total;
}
```

As soon as the `return` statement is met, `calculate_bill` stops executing and returns the value `total`.

A function that returns a value must have a `return` statement. Forgetting it can ruin a program. For instance if `calculate_bill` had read as follows, then the variable `bill` would have had no meaningful value assigned to it, and you might have received a warning from the compiler as well. (The word `void` below indicates that the function does not return a value. In ANSI C, you must place it before the name of any such function.)

```
void calculate_bill (int a, int b, int c)
{
  int total;

  total = a + b + c;
}
```

On the other hand, you do not need to actually use a value when a function returns one. For example, the C input/output functions `printf` and `scanf` return values, but the values are rarely used. See ⟨undefined⟩ [files], page ⟨undefined⟩, for more information on these functions.

If we use the first version of the `calculate_bill` function (the one that contains the line `return total;`), the value of the function can simply be discarded. (Of course, the resulting program is not very useful, since it never displays a value to the user!)

```
int main()
{
  calculate_bill (1, 2, 3);
  exit (0);
}
```

4.4 Function prototyping

Functions do not have to return integer values, as in the above examples, but can return almost any type of value, including floating point and character values. (See Chapter 5 [Variables and declarations], page 19, for more information on variable types.)

A function must be declared to return a certain variable type (such as an integer), just as variables must be. (See Chapter 5 [Variables and declarations], page 19, for more information about variable types.) To write code in good C style, you should declare what type of value a function returns (and what type of parameters it accepts) in two places:

1. At the beginning of the program, in global scope. (See Chapter 6 [Scope], page 27.)
2. In the definition of the function itself.

Function declarations at the beginning of a program are called *prototypes*. Here is an example of a program in which prototypes are used:

```
#include <stdio.h>

void print_stuff (int foo, int bar);
int calc_value (int bas, int quux);


void print_stuff (int foo, int bar)
{
  int var_to_print;

  var_to_print = calc_value (foo, bar);
  printf ("var_to_print = %d\n", var_to_print);
}


int calc_value (int bas, int quux)
{
  return bas * quux;
}


int main()
{
  print_stuff (23, 5);
  exit (0);
}
```

The above program will print the text '`var_to_print = 115`' and then quit.

Prototypes may seem to be a nuisance, but they overcome a problem intrinsic to compilers, which is that they compile functions as they come upon them. Without function prototypes, you usually cannot write code that calls a function before the function itself is defined in the program. If you place prototypes for your functions in a header file, however, you can call the functions from any source code file that includes the header. This is one reason C is considered to be such a flexible programming language.

Some compilers avoid the use of prototypes by making a first pass just to see what functions are there, and a second pass to do the work, but this takes about twice as long. Programmers already hate the time compilers take, and do not want to use compilers that make unnecessary passes on their source code, making prototypes a necessity. Also, prototypes enable the C compiler to do more rigorous error checking, and that saves an enormous amount of time and grief.

4.5 The `exit` function

GNU coding standards specify that you should always use `exit` (or `return`) within your `main` function. (See Chapter 22 [Style], page 219.)

You can use the `exit` function to terminate a program at any point, no matter how many function calls have been made. Before it terminates the program, it calls a number of other functions that perform tidy-up duties such as closing open files.

`exit` is called with a *return code*, like this:

```
exit(0);
```

In the example above, the return code is `0`. Any program that calls your program can read the return code from your program. The return code is like a return value from another function that is not `main`; in fact, most of the time you can use the `return` command within your `main`, instead of `exit`.

Conventionally, a return code of `0` specifies that your program has ended normally and all is well. (You can remember this as "zero errors", although for technical reasons, you cannot use the number of errors your program found as the return code. See Chapter 22 [Style], page 219.) A return code other than `0` indicates that some sort of error has occurred. If your code terminates when it encounters an error, use `exit`, and specify a non-zero return code.

4.6 Questions for Chapter 4

1. Write a function that takes two values a and b, then returns the value of $a * b$ (that is, `a` times `b`.)
2. Is there anything wrong with a function that returns no value?
3. What happens if a function returns a value but you do not assign that value to anything?
4. What happens if a variable is assigned the result of a function, but the function does not return a value?
5. How can you make a program terminate, anywhere in the program?

5 Variables and declarations

Storing data. Discriminating types. Declaring data.

Variable names in C follow the same rules as function names, as far as what characters they can contain. (See Section 4.1 [Function names], page 13.) Variables work differently from functions, however. Every variable in C has a *data type*, or *type*, that conveys to the the compiler what sort of data will be stored in it. Functions in C are sometimes said to have types, but a function's type is actually the data type of the variable it returns.

In some older computer languages like BASIC, and even some newer ones like Perl, you can tell what type a variable is because its name begins or ends with a special character. For example, in many versions of BASIC, all integer variable names end with a percent sign ('`%`') — for example, '`YEAR%`'. No such convention exists in C. Instead, we declare variables, or tell the compiler that they are of a certain type, before they are used. This feature of C has the following advantages (among others):

- It gives a compiler precise information about the amount of memory that will have to be allotted to a variable when a program is run, and what sort of arithmetic will have to be used with it (e.g. integer, floating point, or none at all).
- It provides the compiler with a list of the variables so that it can catch errors in the code, such as assigning a string to an integer variable.

There are a lot of variable types in C. In fact, you can define your own, but there are some basic types ready for use. We will discuss them in the following sections.

5.1 Integer variables

C has five kinds of *integer*. An integer is a whole number (a number without a fractional part). In C, there are a limited number of integers possible; how many depends on the type of integer. In arithmetic, you can have as large a number as you like, but C integer types always have a largest (and smallest) possible number.

- `char`: A single byte, usually one ASCII character. (See the section on the `char` type below.)
- `short`: A short integer (16 bits long on most GNU systems). Also called `short int`. Rarely used.
- `int`: A standard integer (32 bits long on most GNU systems).
- `long`: A long integer (32 bits long on most GNU systems, the same as `int`). Also called `long int`.
- `long long`: A long long integer (64 bits long on most GNU systems). Also called `long long int`.

64-bit operating systems are now appearing in which long integers are 64 bits long. With GCC, long integers are normally 32 bits long and long long integers are 64 bits long, but it varies with the computer hardware and implementation of GCC, so check your system's documentation.

These integer types differ in the size of the integer they can hold and the amount of storage required for them. The sizes of these variables depend on the hardware and operating system of the computer. On a typical 32-bit GNU system, the sizes of the integer types are as follows.

Type	Bits	Possible Values
`char`	8	-127 to 127
`unsigned char`	8	0 to 255
`short`	16	-32,767 to 32,767
`unsigned short`	16	0 to 65,535
`int`	32	-2,147,483,647 to 2,147,483,647
`unsigned int`	32	0 to 4,294,967,295
`long`	32	-2,147,483,647 to 2,147,483,647
`unsigned long`	32	0 to 4,294,967,295
`long long`	64	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
`unsigned long long`	64	0 to 18,446,744,073,709,551,615

On some computers, the lowest possible value may be 1 less than shown here; for example, the smallest possible `short` may be -32,768 rather than -32,767.

The word `unsigned`, when placed in front of integer types, means that only positive or zero values can be used in that variable (i.e. it cannot have a minus sign). The advantage is that larger numbers can then be stored in the same variable. The ANSI standard also allows the word `signed` to be placed before an integer, to indicate the opposite of `unsigned`.

5.1.1 The `char` type

`char` is a special integer type designed for storing single characters. The integer value of a `char` corresponds to an ASCII character. For example, a value of 65 corresponds to the letter '`A`', 66 corresponds to '`B`', 67 to '`C`', and so on.

As in the table above, `unsigned char` permits values from 0 to 255, and `signed char` permits values from -127 (or -128) to 127. The `char` type is signed by default on some computers, but unsigned on others. (See Appendix E [Character conversion table], page 249. See Appendix D [Special characters], page 247.)

`char` is used only within arrays; variables meant to hold one character should be declared `int`. (See Chapter 15 [Strings], page 101, for more information on character arrays. See Section 5.4.1 [Cast operator demo], page 23, for an example of how to use an integer variable to hold a character value.)

5.1.2 Floating point variables

Floating point numbers are numbers with a decimal point. There are different sizes of floating point numbers in C. The `float` type can contain large floating point numbers with a small degree of precision, but the double-precision `double` type can hold even larger numbers with a higher degree of precision. (*Precision* is simply the number of decimal places to which a number can be computed with accuracy. If a number can be computed to five decimal places, it is said to have five *significant digits*.)

All floating point mathematical functions built into C require `double` or `long float` arguments (`long float` variables are generally the same as `double` variables on GNU systems), so it is common to use `float` only for storage of small floating point numbers, and to use `double` everywhere else.

Here are the floating point variable types available in C:

- `float`: A single-precision floating point number, with at least 6 significant decimal digits.
- `double`: A double-precision floating point number. Usually the same as `long float` on GNU systems. Has at least 10 significant decimal digits.
- `long double`: Usually the same as `double` on GNU systems, but may be a 128-bit number in some cases.

On a typical 32-bit GNU system, the sizes of the different floating point types are as follows.

Type	**Bits**	**Possible values (approx.)**
`float`	32	1e-38 to 1e+38
`double`	64	2e-308 to 1e+308
`long double`	64	2e-308 to 1e+308

You may find the figures in the right-hand column confusing. They use a form of shorthand for large numbers. For example, the number 5e2 means $5 * 10\hat{}2$, or 500. 5e-2 means $5 * 10\hat{} - 2$ (5/100, or 1/20). You can see, therefore, that the `float`, `double`, and `long double` types can contain some very large and very small numbers indeed. (When you work with large and small numbers in C, you will use this notation in your code.)

5.2 Declarations

To declare a variable, write the type followed by a list of variables of that type:

type_name variable_name_1, ..., *variable_name_n*;

For example:

```
int last_year, cur_year;
long double earth_mass, mars_mass, venus_mass;
unsigned int num_pets;

long city_pop, state_pop;
state_pop = city_pop = 5000000;

short moon_landing = 1969;
```

```
float temp1, temp2, temp3;
temp1 = 98.6;
temp2 = 98.7;
temp3 = 98.5;

double bignum, smallnum;
bignum = 2.36e208;
smallnum = 3.2e-300;
```

Always declare your variables. A compiler will catch a missing declaration every time and terminate compilation, complaining bitterly. (You will often see a host of error messages, one for each use of the undeclared variable. See Chapter 23 [Debugging], page 223.)

5.3 Initialization

Assigning a variable its first value is called *initializing* the variable. When you declare a variable in C, you can also initialize it at the same time. This is no more efficient in terms of a running program than doing it in two stages, but sometimes creates tidier and more compact code. Consider the following:

```
int initial_year;
float percent_complete;

initial_year = 1969;
percent_complete = 89.5;
```

The code above is equivalent to the code below, but the code below is more compact.

```
int initial_year = 1969;
float percent_complete = 89.5;
```

You can always write declarations and initializers this way, but you may not always want to. (See Chapter 22 [Style], page 219.)

5.4 The cast operator

An *operator* is a symbol or string of C characters used as a function. One very valuable operator in C is the *cast operator*, which converts one type into another. Its general form is as follows:

(*type*) *variable*

For example, floating point and integer types can be interconverted:

```
float exact_length;
int rough_length;

exact_length = 3.37;
rough_length = (int) exact_length;
```

In the example above, the cast operator rounds the number down when converting it from a float to an integer, because an integer number cannot represent the fractional part after the decimal point. Note that C always *truncates*, or rounds down, a number when converting it to an integer. For example, both 3.1 and 3.9 are truncated to 3 when C is converting them to integer values.

The cast operator works the other way around, too:

```
float exact_length;
int rough_length;

rough_length = 12;
exact_length = (float) rough_length;
```

In converting large integers to floating point numbers, you may lose some precision, since the `float` type guarantees only 6 significant digits, and the `double` type guarantees only 10.

It does not always make sense to convert types. (See Chapter 20 [Data structures], page 197, for examples of types that do not convert to other types well.)

5.4.1 Cast operator demo

The following is an example of how to use the cast operator in C code. It also shows how to use an integer variable to store a character value.

```
/***************************************************/
/*                                                 */
/* Demo of Cast operator                           */
/*                                                 */
/***************************************************/

#include <stdio.h>

int main()               /* Use int float and int */
{
  float my_float;
  int my_int;
  int my_ch;

  my_float = 75.345;
  my_int = (int) my_float;
  my_ch = (int) my_float;
  printf ("Convert from float my_float=%f to my_int=%d and my_ch=%c\n",
    my_float, my_int, my_ch);

  my_int = 69;
  my_float = (float) my_int;
  my_ch = my_int;
  printf ("Convert from int my_int=%d to my_float=%f and my_ch=%c\n",
    my_int, my_float, my_ch);

  my_ch = '*';
  my_int = my_ch;
  my_float = (float) my_ch;
  printf ("Convert from int my_ch=%c to my_int=%d and my_float=%f\n",
    my_ch, my_int, my_float);
```

```
  exit(0);
}
```

Here is the sort of output you should expect (floating point values may differ slightly):

```
Convert from float my_float=75.345001 to my_int=75 and my_ch=K
Convert from int my_int=69 to my_float=69.000000 and my_ch=E
Convert from int my_ch=* to my_int=42 and my_float=42.000000
```

5.5 Storage classes

There are a few variable declaration keywords commonly used in C that do not specify variable types, but a related concept called *storage classes*. Two common examples of storage class specifiers are the keywords `extern` and `static`.

5.5.1 External variables

Sometimes the source code for a C program is contained in more than one text file. If this is the case, then it may be necessary to use variables that are defined in another file. You can use a global variable in files other than the one in which it is defined by redeclaring it, prefixed by the `extern` specifier, in the other files.

File main.c

```
int main()
{
  extern int my_var;

  my_var = 500;
  print_value();
  exit (0);
}
```

File secondary.c

```
#include <stdio.h>
int my_var;

void print_value()
{
  printf("my_var = %d\n", my_var);
}
```

In this example, the variable `my_var` is created in the file '`secondary.c`', assigned a value in the file '`main.c`', and printed out in the function `print_value`, which is defined in the file '`secondary.c`', but called from the file '`main.c`'.

See Section 17.4 [Compiling multiple files], page 166, for information on how to compile a program whose source code is split among multiple files. For this example, you can simply type the command `gcc -o testprog main.c secondary.c`, and run the program with `./testprog`.

5.5.2 Static variables

A second important storage class specifier is `static`. Normally, when you call a function, all its local variables are reinitialized each time the function is called. This means that their values change between function calls. Static variables, however, maintain their value between function calls.

Every global variable is defined as `static` automatically. (Roughly speaking, functions anywhere in a program can refer to a global variable; in contrast, a function can only refer to a local variable that is "nearby", where "nearby" is defined in a specific manner. See Chapter 6 [Scope], page 27, for more information on global variables. See Chapter 7 [Expressions and operators], page 31, for an example of a static local variable.)

5.5.3 Other storage classes

There are three more storage class identifiers in C: `auto`, `register`, and `typedef`.

- `auto` is the opposite of `static`. It is redundant, but is included in contemporary versions of C for backwards compatibility. All local variables are `auto` by default.
- `register` is another outdated C storage class. Defining a variable as `register` used to store it in one of the computer's registers, a specific location on its processor chip, thereby making code using that variable run faster. These days, most C compilers (including GCC) are smart enough to *optimize* the code (make it faster and more compact) without the `register` keyword.
- `typedef` allows you to define your own variable types. See Chapter 19 [More data types], page 189, for more information.

5.6 Questions for Chapter 5

1. What is an identifier?
2. Which of the following are valid C variable names?
    1. `Ralph23`
    2. `80shillings`
    3. `mission_control`
    4. `A%`
    5. `A$`
    6. `_off`
3. Write a statement to declare two integers called `start_temperature` and `end_temperature`.
4. What is the difference between the types `float` and `double`?
5. What is the difference between the types `int` and `unsigned int`?
6. Write a statement that assigns the value 1066 to the integer variable `norman`.
7. What data type do C functions return by default?
8. You must declare the data type a function returns at two places in a program. Where?
9. Write a statement, using the cast operator, to print out the integer part of the number 23.1256.
10. Is it possible to have an automatic global variable?

6 Scope

Where a program's fingers can and can't reach.

Imagine that a function is a building with a person (Fred) standing in the doorway. This person can see certain things: other people and other buildings, out in the open. But Fred cannot see certain other things, such as the people inside the other buildings. Just so, some variables in a C program, like the people standing outside, are *visible* to nearly every other part of the program (these are called *global variables*), while other variables, like the people indoors, are hidden behind the "brick walls" of curly brackets (these are called *local variables*).

Where a variable is visible to other C code is called the *scope* of that variable. There are two main kinds of scope, global and local, which stem from the two kinds of places in which you can declare a variable:

1. *Global scope* is outside all of the functions, that is, in the space between function definitions — after the `#include` lines, for example. Variables declared in global scope are called *global variables*. Global variables can be used in any function, as well as in any block within that function.

```
#include <stdio.h>

int global_integer;
float global_floating_point;

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

2. You can also declare variables immediately following the opening bracket ('`{`') of any block of code. This area is called *local scope*, and variables declared here are called *local variables*. A local variable is visible within its own block and the ones that block contains, but invisible outside its own block.

```
#include <stdio.h>

int main()
{
  int foo;
  float bar, bas, quux;

  exit (0);
}
```

6.1 Global Variables

Global variables can be used in any function, as well as any block within that function. (Technically, global variables can only be seen by functions that are defined after the declaration of those global variables, but global variables are usually declared in a header file that is included everywhere they are needed.) Global variables are created when a program is started and are not destroyed until a program is stopped.

6.2 Local Variables

Local variables, on the other hand, are only visible within local scope. They are “trapped” inside their code blocks.

Just as global scope contains many functions, however, each function can contain many code blocks (defined with curly brackets: ‘{...}’). C allows blocks within blocks, even functions within functions, *ad infinitum*. A local variable is visible within its own block and the ones that block contains, but invisible outside its own block.

```
int a;

/* Global scope.  Global variable 'a' is visible here,
   but not local variables 'b' or 'c'. */

int main()
{
  int b;

  /* Local scope of 'main'.
     Variables 'a' and 'b' are visible here,
     but not 'c'. */

  {
    int c;

    /* Local scope of `{...}' block within 'main'.
       Variables 'a', 'b', and 'c' are all visible here. */
  }

  exit (0);
}
```

Local variables are not visible outside their curly brackets. To use an “existence” rather than a “visibility” metaphor, local variables are created when the opening brace is met, and they are destroyed when the closing brace is met. (Do not take this too literally; they are not created and destroyed in your C source code, but internally to the computer, when you run the program.)

6.3 Communication via parameters

If no code inside a function could ever communicate with other parts of the program, then functions would not be very useful. Functions would be isolated, comatose, unable to do much of anything. Fortunately, although local variables are invisible outside their code blocks, they can still communicate with other functions via parameters. See Chapter 7 [Expressions and operators], page 31, the next chapter, for information on parameters.

6.4 Scope example

Notice that there are two variables named `my_var` in the example below, both visible in the same place. When two or more variables visible in one area of code have the same name, the last variable to be defined takes priority. (Technically adept readers will realize that this is because it was the last one onto the variable stack.)

```
/**************************************************************/
/*                                                            */
/* SCOPE                                                      */
/*                                                            */
/**************************************************************/

#include <stdio.h>

int main ()
{
  int my_var = 3;

  {
    int my_var = 5;
    printf ("my_var=%d\n", my_var);
  }

  printf ("my_var=%d\n", my_var);

  exit(0);
}
```

When you run this example, it will print out the following text:

```
my_var=5
my_var=3
```

6.5 Questions for Chapter 6

1. What is a global variable?
2. What is a local variable?
3. Do parameters spoil functions by leaking the variables into other functions?
4. Write a program `gnoahs_park` that declares 4 variables. Two *global* integer variables called `num_gnus` and `num_gnats`, and two *local* floating point variables within the function `main`, called `avg_gnu_mass`, and `avg_gnat_mass`. Then add another function called `calculate_park_biomass`, and pass `avg_gnu_mass` and `avg_gnat_mass` to it. How many different storage spaces are used when this program runs? (Hint: are `avg_gnu_mass` and `avg_gnat_mass` and their copies the same?)

7 Expressions and operators

Thinking in C. Short strings of code.

An *operator* is a character or string of characters used as a built-in function. We have already experimented with one operator in C: the cast operator.

An operator is so called because it takes one or more values and *operates* on them to produce a result. For example, the addition operator + can operate on the values 4 and 5 to produce the result 9. Such a procedure is called an *operation*, and any value operated on (such as 4 and 5 in this example) is called an *operand*.

There are many operators in C. Some of them are familiar, such as the addition operator + and subtraction operator -. Most operators can be thought of as belonging to one of three groups, according to what they do with their operands:

- Mathematical operators, such as the addition operator + in `100 + 500`, or the multiplication operator * in `12 * 2`.
- Comparison operators (a subset of mathematical operators), such as the less-than operator < and the greater-than operator >.
- Operators that produce new variable types, such as the cast operator.

The majority of operators fall into the first group. The second group is a subset of the first set; in this second set, the result of an operation is a *Boolean value* (a value of either true or false).

C has about forty different operators. The chief object of this chapter is to explain the basic operators in C. We will examine more complex operators in another chapter. (See Chapter 18 [Advanced operators], page 177.)

7.1 The assignment operator

No operator such as addition (+) or multiplication (*) would be useful without another operator that attaches the values they produce to variables. Thus, the assignment operator = is perhaps the most important mathematical operator.

We have seen the assignment operator already in our code examples. Here is an example to refresh your memory:

```
int gnu_count, gnat_count, critter_count;

gnu_count = 45;
gnat_count = 5678;

critter_count = gnu_count + gnat_count;
```

The assignment operator takes the value of whatever is on the right-hand side of the = symbol and puts it into the variable on the left-hand side. For example, the code sample above assigns the value 45 to the variable `gnu_count`.

Something that can be assigned *to* is called an *lvalue*, ("l" for "left", because it can appear on the *left* side of an assignment). You will sometimes see the word '`lvalue`' in error messages from the compiler. For example, try to compile a program containing the following code:

```
5 = 2 + 3;
```

You will receive an error such as the following:

error

```
bad_example.c:3: invalid lvalue in assignment
```

You can't assign a value to 5; it has its own value already! In other words, 5 is not an lvalue.

7.1.1 Important note about assignment

Many people confuse the assignment operator (`=`) with the equality operator (`==`), and this is a major source of bugs in C programs. Because of early arithmetic training, people tend to think of `=` as indicating equality, but in C it means "takes on the value produced by", and it should always be read that way. By way of contrast, `==` is an equality test operator and should always be read "is tested for equality with". (See Section 7.8 [Comparisons and logic], page 36, for more information on the `==` operator.)

7.2 Expressions and values

The most common operators in any language are basic arithmetic operators. In C, these include the following:

`+` unary plus, example: `+5`

`-` unary minus, example: `-5`

`+` addition, example: `2 + 2`

`-` subtraction, example: `14 - 7`

`*` multiplication, example: `3 * 3`

`/` floating point division, example: `10.195 / 2.4`

`/` integer division *div*, example: `5 / 2`

`%` integer remainder *mod*, example: `24 % 7`

7.3 Expressions

An *expression* is simply a string of operators, variables, numbers, or some combination, that can be parsed by the compiler. All of the following are expressions:

```
19

1 + 2 + 3

my_var

my_var + some_function()

(my_var + 4 * (some_function() + 2))
```

```
32 * circumference / 3.14

day_of_month % 7
```

Here is an example of some arithmetic expressions in C:

```
#include <stdio.h>

int main ()
{
  int my_int;

  printf ("Arithmetic Operators:\n\n");

  my_int = 6;
  printf ("my_int = %d, -my_int = %d\n", my_int, -my_int);

  printf ("int 1 + 2 = %d\n", 1 + 2);
  printf ("int 5 - 1 = %d\n", 5 - 1);
  printf ("int 5 * 2 = %d\n", 5 * 2);

  printf ("\n9 div 4 = 2 remainder 1:\n");
  printf ("int 9 / 4 = %d\n", 9 / 4);
  printf ("int 9 % 4 = %d\n", 9 % 4);

  printf ("double 9 / 4 = %f\n", 9.0 / 4.0);

  return 0;
}
```

The program above produces the output below:

```
Arithmetic Operators:

my_int = 6, -my_int = -6
int 1 + 2 = 3
int 5 - 1 = 4
int 5 * 2 = 10

9 div 4 = 2 remainder 1:
int 9 / 4 = 2
int 9 % 4 = 1
double 9 / 4 = 2.250000
```

7.4 Parentheses and Priority

Just as in algebra, the C compiler considers operators to have certain priorities, and *evaluates*, or parses, some operators before others. The order in which operators are evaluated is called *operator precedence* or the *order of operations*. You can think of some operators as "stronger" than others. The "stronger" ones will always be evaluated first; otherwise, expressions are evaluated from left to right.

For example, since the multiplication operator `*` has a higher priority than the addition operator `+` and is therefore evaluated first, the following expression will always evaluate to 10 rather than 18:

```
4 + 2 * 3
```

However, as in algebra, you can use parentheses to force the program to evaluate the expression to 18:

```
(4 + 2) * 3
```

The parentheses force the expression `(4 + 2)` to be evaluated first. Placing parentheses around `2 * 3`, however, would have no effect.

Parentheses are classed as operators by the compiler; they have a value, in the sense that they assume the value of whatever is inside them. For example, the value of `(5 + 5)` is 10.

7.5 Unary Operator Precedence

Unary operators are operators that have only a single operand — that is, they operate on only one object. The following are (or can be) all unary operators:

```
++  --  +  -
```

The order of evaluation of unary operators is from right to left, so an expression like:

```
*ptr++;
```

would perform the `++` before the `*`. (The `++` operator will be introduced in the next section, and the `*` operator will be introduced in the next chapter. See Chapter 9 [Pointers], page 45.)

7.6 Special Assignment Operators ++ and --

C has some special operators that can simplify code. The simplest of these are the increment and decrement operators:

`++` increment: add one to

`--` decrement: subtract one from

You can use these with any integer or floating point variable (or a character in some cases, carefully). They simply add or subtract 1 from a variable. The following three statements are equivalent:

```
variable = variable + 1;
variable++;
++variable;
```

So are these three:

```
variable = variable - 1;
variable--;
--variable;
```

Notice that the `++` and `--` operators can be placed before or after the variable. In the cases above, the two forms work identically, but there is actually a subtle difference. (See Section 18.1.2 [Postfix and prefix ++ and –], page 179, for more information.)

7.7 More Special Assignments

Like ++ and --, the following operators are short ways of writing longer expressions. Consider the following statement:

```
variable = variable + 23;
```

In C, this would be a long-winded way of adding 23 to `variable`. It could be done more simply with the general increment operator +=, as in this example:

```
variable += 23;
```

This performs exactly the same operation. Similarly, the following two statements are equivalent:

```
variable1 = variable1 + variable2;
variable1 += variable2;
```

There are a handful of these operators. For example, one for subtraction:

```
variable = variable - 42;
variable -= 42;
```

More surprisingly, perhaps, there is one for multiplication:

```
variable = variable * 2;
variable *= 2;
```

The main arithmetic operators all follow this pattern:

`+=` addition assignment operator

`-=` subtraction assignment operator

`*=` multiplication assignment operator

`/=` division assignment operator (floating point and integers)

`%=` remainder assignment operator (integers only)

There are more exotic kinds too, used for machine-level operations, which we will ignore for the moment. (See Chapter 18 [Advanced operators], page 177, if you want to know more.)

Here is a short program that demonstrates these special assignment operators:

```
#include <stdio.h>

int main()
{
  int my_int;

  printf ("Assignment Operators:\n\n");

  my_int = 10;                          /* Assignment */
  printf ("my_int = 10 : %d\n",my_int);

  my_int++;                             /* my_int = my_int + 1 */
  printf ("my_int++    : %d\n",my_int);

  my_int += 5;                          /* my_int = my_int + 5 */
```

```
    printf ("my_int += 5 : %d\n",my_int);

    my_int--;                              /* my_int = my_int = 1 */
    printf ("my_int--    : %d\n",my_int);

    my_int -= 2;                           /* my_int = my_int - 2 */
    printf ("my_int -= 2 : %d\n",my_int);

    my_int *= 5;                           /* my_int = my_int * 5 */
    printf ("my_int *= 5 : %d\n",my_int);

    my_int /= 2;                           /* my_int = my_int / 2 */
    printf ("my_int /= 2 : %d\n",my_int);

    my_int %= 3;                           /* my_int = my_int % 3 */
    printf ("my_int %%= 3 : %d\n",my_int);

    return 0;
  }
```

The program above produces the output below:

```
Assignment Operators:

my_int = 10 : 10
my_int++    : 11
my_int += 5 : 16
my_int--    : 15
my_int -= 2 : 13
my_int *= 5 : 65
my_int /= 2 : 32
my_int %= 3 : 2
```

The second to last line of output is

```
my_int /= 2 : 32
```

In this example, 65 divided by 2 using the `/=` operator results in 32, not 32.5. This is because both operands, 65 and 2, are integers, type `int`, and when `/=` operates on two integers, it produces an integer result. This example only uses integer values, since that is how the numbers are declared. To get the fractional answer, you would have had to declare the three numbers involved as floats.

The last line of output is

```
my_int %= 3 : 2
```

This is because 32 divided by 3 is 10 with a remainder of 2.

7.8 Comparisons and logic

Comparison operators tell you how numerical values relate to one another, such as whether they are equal to one another, or whether one is greater than the other. Comparison operators are used in logical tests, such as `if` statements. (See Chapter 10 [Decisions], page 53.)

The results of a logical comparison are always either true (1) or false (0). In computer programming jargon, true and false are the two *Boolean values*. Note that, unlike real life, there are no "gray areas" in C; just as in Aristotelian logic, a comparison operator will never produce a value other than true or false.

Six operators in C are used to make logical comparisons:

`==` is equal to

`!=` is not equal to

`>` is greater than

`<` is less than

`>=` is greater than or equal to

`<=` is less than or equal to

Important: Remember that many people confuse the equality operator (`==`) with the assignment operator (`=`), and this is a major source of bugs in C programs. (See Section 7.2 [Expressions and values], page 32, for more information on the distinction between the `==` and `=` operators.)

The operators above result in values, much as the addition operator `+` does. They produce Boolean values: true and false only. Actually, C uses 1 for "true" and 0 for "false" when evaluating expressions containing comparison operators, but it is easy to define the strings '`TRUE`' and '`FALSE`' as macros, and they may well already be defined in a library file you are using. (See Chapter 12 [Preprocessor directives], page 71, for information on defining macros.)

```
#define TRUE  1
#define FALSE 0
```

Note that although any non-zero value in C is treated as true, you do not need to worry about a comparison evaluating to anything other than 1 or 0. Try the following short program:

```
#include <stdio.h>

int main ()
{
  int truth, falsehood;

  truth = (2 + 2 == 4);
  falsehood = (2 + 2 == 5);

  printf("truth is %d\n", truth);
  printf("falsehood is %d\n", falsehood);

  exit (0);
}
```

You should receive the following result:

```
truth is 1
falsehood is 0
```

7.9 Logical operators

Comparisons are often made in pairs or groups. For example, you might want to ask a question such as, "Is variable `a` greater than variable `b` *and* is variable `b` greater than variable `c`?" The word "and" in the question above is represented in C by the *logical operator* (an "operator on Boolean values") `&&`, and the whole comparison above might be represented by the following expression:

```
(a > b) && (b > c)
```

The main logical operators in C are as follows:

`&&` logical AND

`||` logical Inclusive OR (See Section 7.9.1 [Inclusive OR], page 38.)

`!` logical NOT

Here is another example. The question, "Is the variable `a` greater than the variable `b`, *or* is the variable `a` *not* greater than the variable `c`?" might be written:

```
(a > b) || !(a > c)
```

7.9.1 Inclusive OR

Note well! Shakespeare might have been disappointed that, *whatever* the value of a variable `to_be`, the result of

```
to_be || !to_be
```

(i.e. "To be, or not to be?") is always 1, or true. This is because one or the other of `to_be` or `!to_be` must always be true, and as long as one side of an OR `||` expression is true, the whole expression is true.

7.10 Questions for Chapter 7

1. What is an operand?
2. Write a short statement that assigns the remainder of 5 divided by 2 to a variable called `remainder` and prints it out.
3. Write a statement that subtracts -5 from 10.

8 Parameters

Ways in and out of functions.

Parameters are the main way in C to transfer, or *pass*, information from function to function. Consider a call to our old friend `calculate_bill`:

```
total = calculate_bill (20, 35, 28);
```

We are passing 20, 35, and 28 as parameters to `calculate_bill` so that it can add them together and return the sum.

When you pass information to a function with parameters, in some cases the information can go only one way, and the function returns only a single value (such as `total` in the above snippet of code). In other cases, the information in the parameters can go both ways; that is, the function called can alter the information in the parameters it is passed.

The former technique (passing information only one way) is called *passing parameters by value* in computer programming jargon, and the latter technique (passing information both ways) is referred to as *passing parameters by reference*.

For our purposes, at the moment, there are two (mutually exclusive) kinds of parameters:

- *Value parameters* are the kind that pass information one-way. They are so-called because the function to which they are passed receives only a copy of their values, and they cannot be altered as variable parameters can. The phrase "passing by value" mentioned above is another way to talk about passing "value parameters".
- *Variable parameters* are the kind that pass information back to the calling function. They are so called because the function to which they are passed can alter them, just as it can alter an ordinary variable. The phrase "passing by reference" mentioned above is another way to talk about passing "variable parameters".

Consider a slightly-expanded version of `calculate_bill`:

```
#include <stdio.h>

int main (void);
int calculate_bill (int, int, int);

int main()
{
  int bill;
  int fred = 25;
  int frank = 32;
  int franny = 27;

  bill = calculate_bill (fred, frank, franny);
  printf("The total bill comes to $%d.00.\n", bill);

  exit (0);
}

int calculate_bill (int diner1, int diner2, int diner3)
{
```

```
  int total;

  total = diner1 + diner2 + diner3;
  return total;
}
```

Note that all of the parameters in this example are value parameters: the information flows only one way. The values are passed to the function `calculate_bill`. The original values are not changed. In slightly different jargon, we are "passing the parameters by value only". We are *not* passing them "by reference"; they are *not* "variable parameters".

All parameters must have their types declared. This is true whether they are value parameters or variable parameters. In the function `calculate_bill` above, the value parameters `diner1`, `diner2`, and `diner3` are all declared to be of type `int`.

8.1 Parameters in function prototypes

Note that in the function prototype for `calculate_bill`, the parameter names were completely omitted. This is perfectly acceptable in ANSI C, although it might be confusing to someone trying to understand your code by reading the function prototypes, which can be in a separate file from the functions themselves. For instance, in the code example above, the function prototype for `calculate_bill` looks like this:

```
int calculate_bill (int, int, int);
```

You may include parameter names in function prototypes if you wish; this is usually a good idea when the function prototype is significantly separated from the function definition, such as when the prototype is in a header file or at the top of a long file of function definitions. For example, we could have written the prototype for `calculate_bill` thus:

```
int calculate_bill (int diner1, int diner2, int diner3);
```

Parameter names in a function prototype do not need to match the names in the function's definition; only their types need to match. For example, we can also write the function prototype above in this way:

```
int calculate_bill (int guest1, int guest2, int guest3);
```

As usual, it is a good idea to use mnemonic names for the parameters in a function prototype, as in the last two examples.[1] Thus, the function prototype below is not as helpful to the person reading your code as the last two examples are; it might just as well have been written without variable names at all:

```
int calculate_bill (int variable1, int variable2, int variable3);
```

8.2 Value Parameters

When you are passing data to a function by value, the parameters in the function you are passing the data *to* contain copies of the data in the parameters you are passing the data *with*. Let us modify the function `main` from the last example slightly:

[1] That is, unless you are competing in The International Obfuscated C Code Contest (`http://www.ioccc.org/`).

```
int main()
{
  int bill;
  int fred = 25;
  int frank = 32;
  int franny = 27;

  bill = calculate_bill (fred, frank, franny);

  fred = 20000;
  frank = 50000;
  franny = 20000;

  printf("The total bill comes to $%d.00.\n", bill);

  exit (0);
}
```

As far as the function `calculate_bill` is concerned, `fred`, `frank`, and `franny` are still 25, 32, and 27 respectively. Changing their values to extortionate sums after passing them to `calculate_bill` does nothing; `calculate_bill` has already created local copies of the parameters, called `diner1`, `diner2`, and `diner3` containing the earlier values.

Important: Even if we named the parameters in the definition of `calculate_bill` to match the parameters of the function call in `main` (see example below), the result would be the same: `main` would print out '`$84.00`', not '`$90000.00`'. When passing data by value, the parameters in the function call and the parameters in the function definition (which are only copies of the parameters in the function call) are completely separate.

Just to remind you, this is the `calculate_bill` function:

```
int calculate_bill (int fred, int frank, int franny)
{
  int total;

  total = fred + frank + franny;
  return total;
}
```

8.3 Actual parameters and formal parameters

There are two other categories that you should know about that are also referred to as "parameters". They are called "parameters" because they define information that is passed to a function.

- *Actual parameters* are parameters as they appear in function calls.
- *Formal parameters* are parameters as they appear in function declarations.

A parameter cannot be both a formal and an actual parameter, but both formal parameters and actual parameters can be either value parameters or variable parameters.

Let's look at `calculate_bill` again:

```
#include <stdio.h>

int main (void);
int calculate_bill (int, int, int);

int main()
{
  int bill;
  int fred = 25;
  int frank = 32;
  int franny = 27;

  bill = calculate_bill (fred, frank, franny);
  printf("The total bill comes to $%d.00.\n", bill);

  exit (0);
}

int calculate_bill (int diner1, int diner2, int diner3)
{
  int total;

  total = diner1 + diner2 + diner3;
  return total;
}
```

In the function `main` in the example above, `fred`, `frank`, and `franny` are all actual parameters when used to call `calculate_bill`. On the other hand, the corresponding variables in `calculate_bill` (namely `diner1`, `diner2` and `diner3`, respectively) are all formal parameters because they appear in a function definition.

Although formal parameters are always variables (which does not mean that they are always variable parameters), actual parameters do not have to be variables. You can use numbers, expressions, or even function calls as actual parameters. Here are some examples of valid actual parameters in the function call to `calculate_bill`:

```
bill = calculate_bill (25, 32, 27);

bill = calculate_bill (50+60, 25*2, 100-75);

bill = calculate_bill (fred, franny, (int) sqrt(25));
```

(The last example requires the inclusion of the math routines in '`math.h`', and compilation with the '`-lm`' option. `sqrt` is the square-root function and returns a `double`, so it must be cast into an `int` to be passed to `calculate_bill`.)

8.4 Variadic functions

Suppose you are writing a program that repeatedly generates lists of numbers that can run anywhere from one to fifty items. You never know how many numbers a particular list will contain, but you always want to add all the numbers together. Passing them to an ordinary C function will not work, because an ordinary function has a fixed number of formal parameters, and cannot accept an arbitrarily long list of actual parameters. What should you do?

One way of solving this problem is to use a *variadic function*, or function that can accept arbitrarily long lists of actual parameters. You can do this by including the '`stdarg.h`' header in your program. For example, with '`stdarg.h`', you can write a function called `add_all` that will add all integers passed to it, returning correct results for all of the following calls:

```
sum = add_all (2, 3, 4);

sum = add_all (10, 150, 9, 81, 14, 2, 2, 31);

sum = add_all (4);
```

Unfortunately, the use of '`stdarg.h`' is beyond the scope of this tutorial. For more information on variadic functions, see the GNU C Library manual (`http://www.gnu.org/manual/glibc-2.0.6/libc.html`).

8.5 Questions for Chapter 8

1. What is the difference between a value parameter and a variable parameter?
2. What is the difference between a formal parameter and an actual parameter?
3. What does passing by reference let you do that passing by value doesn't?
4. Can a function call be used as an actual parameter?
5. Do actual and formal parameters need to have the same names?

9 Pointers

Making maps of data.

In one sense, any variable in C is just a convenient label for a chunk of the computer's memory that contains the variable's data. A *pointer*, then, is a special kind of variable that contains the location or *address* of that chunk of memory. (Pointers are so called because they *point* to a chunk of memory.) The address contained by a pointer is a lengthy number that enables you to pinpoint exactly where in the computer's memory the variable resides.

Pointers are one of the more versatile features of C. There are many good reasons to use them. Knowing a variable's address in memory enables you to pass the variable to a function by reference (See Section 9.4 [Variable parameters], page 49.)[1] Also, since functions are just chunks of code in the computer's memory, and each of them has its own address, you can create pointers to functions too, and knowing a function's address in memory enables you to pass functions as parameters too, giving your functions the ability to switch among calling numerous functions. (See [Function pointers], page 276.)

Pointers are important when using text strings. In C, a text string is always accessed with a pointer to a character — the first character of the text string. For example, the following code will print the text string '`Boy howdy!`':

```
char *greeting = "Boy howdy!";
printf ("%s\n\n", greeting);
```

See Chapter 15 [Strings], page 101.

Pointers are important for more advanced types of data as well. For example, there is a data structure called a "linked list" that uses pointers to "glue" the items in the list together. (See Chapter 20 [Data structures], page 197, for information on linked lists.)

Another use for pointers stems from functions like the C input routine `scanf`. This function accepts information from the keyboard, just as `printf` sends output to the console. However, `scanf` uses pointers to variables, not variables themselves. For example, the following code reads an integer from the keyboard:

```
int my_integer;
scanf ("%d", &my_integer);
```

(See Section 16.2.9.1 [scanf], page 135, for more information.)

9.1 Pointer operators

To create a pointer to a variable, we use the `*` and `&` operators. (In context, these have nothing to do with multiplication or logical AND. For example, the following code declares a variable called `total_cost` and a pointer to it called `total_cost_ptr`.

```
float total_cost;
float *total_cost_ptr;

total_cost_ptr = &total_cost;
```

[1] This, by the way, is how the phrase "pass by reference" entered the jargon. Like other pointers, a variable parameter "makes a reference" to the address of a variable.

The '*' symbol in the declaration of `total_cost_ptr` is the way to declare that variable to be a pointer in C. (The '_ptr' at the end of the variable name, on the other hand, is just a way of reminding humans that the variable is a pointer.)

When you read C code to yourself, it is often useful to be able to pronounce C's operators aloud; you will find it can help you make sense of a difficult piece of code. For example, you can pronounce the above statement `float *total_cost_ptr` as "Declare a float pointer called `total_cost_ptr`", and you can pronounce the statement `total_cost_ptr = &total_cost;` as "Let `total_cost_ptr` take as its value the address of the variable `total_cost`".

Here are some suggestions for pronouncing the `*` and `&` operators, which are always written in front of a variable:

`*` "The contents of the address held in *variable*" or "the contents of the location pointed to by *variable*".

`&` "The address of *variable*" or "the address at which the variable *variable* is stored".

For instance:

`&fred` "The address of `fred`" or "the address at which the variable `fred` is stored".

`*fred_ptr`
"The contents of the address held in `fred_ptr`" or "the contents of the location pointed to by `fred_ptr`".

The following examples show some common ways in which you might use the `*` and `&` operators:

```
int some_var;                          /* 1 */
```
"Declare an integer variable called `some_var`."

```
int *ptr_to_some_var;                  /* 2 */
```
"Declare an integer pointer called `ptr_to_some_var`." (The `*` in front of `ptr_to_some_var` is the way C declares `ptr_to_some_var` as a pointer to an integer, rather than just an integer.)

```
some_var = 42;                         /* 3 */
```
"Let `some_var` take the value 42."

```
ptr_to_some_var = &some_var;             /* 4 */
```
"Let `ptr_to_some_var` take the address of the variable `some_var` as its value." (Notice that only now does `ptr_to_some_var` become a pointer to the particular variable `some_var` — before this, it was merely a pointer that could point to *any* integer variable.)

```
printf ("%d\n\n", *ptr_to_some_var);    /* 5 */
```
"Print out the contents of the location pointed to by `ptr_to_some_var`." (In other words, print out `some_var` itself. This will print just 42. Accessing what a pointer points to in this way is called *dereferencing* the pointer, because the pointer is considered to be *referencing* the variable.)

`*ptr_to_some_var = 56; /* 6 */` "Let the contents of the location pointed to by `ptr_to_some_var` equal 56." (In the context of the other statements, this is the same as the more direct statement `some_var = 56;`.)

A subtle point: don't confuse the usage of asterisks in code like examples 2 and 6 above. Using an asterisk in a declaration, as in example 2, declares the variable to be a pointer, while using it on the left-hand side of an assignment, as in example 6, dereferences a variable that is already a pointer, enabling you to access the variable to which the pointer is pointing.

9.2 Pointer types

Pointers can point to any type of variable, but they must be declared to do so. A pointer to an integer is not the same type of variable as a pointer to a float or other variable type. At the "business end" of a pointer is usually a variable, and all variables have a type.

Here are some examples of different types of pointer:

```
int *my_integer_ptr;

char *my_character_ptr;

float *my_float_ptr;

double *my_double_ptr;
```

However, GCC is fairly lenient about casting different types of pointer to one another *implicitly*, or automatically, without your intervention. For example, the following code will simply truncate the value of `*float_ptr` and print out 23. (As a bonus, pronunciation is given for every significant line of the code in this example.)

```
#include <stdio.h>
/* Include the standard input/output header in this program */

int main()
/* Declare a function called main that returns an integer
   and takes no parameters */
{
```

```
    int *integer_ptr;
    /* Declare an integer pointer called integer_ptr */

    float *float_ptr;
    /* Declare a floating-point pointer called float_ptr */

    int my_int = 17;
    /* Declare an integer variable called my_int
       and assign it the value 17 */

    float my_float = 23.5;
    /* Declare a floating-point variable called my_float
       and assign it the value 23.5 */

    integer_ptr = &my_int;
    /* Assign the address of the integer variable my_int
       to the integer pointer variable integer_ptr */

    float_ptr = &my_float;
    /* Assign the address of the floating-point variable my_float
       to the floating-point pointer variable float_ptr */

    *integer_ptr = *float_ptr;
    /* Assign the contents of the location pointed to by
       the floating-point pointer variable float_ptr
       to the location pointed to by the integer pointer variable
       integer_ptr (the value assigned will be truncated) */

    printf ("%d\n\n", *integer_ptr);
    /* Print the contents of the location pointed to by the
       integer pointer variable integer_ptr */

    return 0;
    /* Return a value of 0, indicating successful execution,
       to the operating system */

  }
```

There will still be times when you will want to convert one type of pointer into another. For example, GCC will give a warning if you try to pass float pointers to a function that accepts integer pointers. Not treating pointer types interchangeably will also help you understand your own code better.

To convert pointer types, use the cast operator. (See Section 5.4 [The cast operator], page 22.) As you know, the general form of the cast operator is as follows:

(*type*) *variable*

Here is the general form of the cast operator for pointers:

(*type* *) *pointer_variable*

Here is an actual example:

```
int *my_integer_ptr;
long *my_long_ptr;

my_long_ptr = (long *) my_integer_ptr;
```

This copies the value of the pointer `my_integer` to the pointer `my_long_ptr`. The cast operator ensures that the data types match. (See Chapter 20 [Data structures], page 197, for more details on pointer casting.)

9.3 Pointers and initialization

You should not initialize pointers with a value when you declare them, although the compiler will not prevent this. Doing so simply makes no sense. For example, think about what happens in the following statement:

```
int *my_int_ptr = 2;
```

First, the program allocates space for a pointer to an integer. Initially, the space will contain *garbage* (random data). It will not contain actual data until the pointer is "pointed at" such data. To cause the pointer to refer to a real variable, you need another statement, such as the following:

```
my_int_ptr = &my_int;
```

On the other hand, if you use just the single initial assignment, `int *my_int_ptr = 2;`, the program will try to fill the contents of the memory location pointed to by `my_int_ptr` with the value 2. Since `my_int_ptr` is filled with garbage, it can be *any* address. This means that the value 2 might be stored anywhere. anywhere, and if it overwrites something important, it may cause the program to crash.

The compiler will warn you against this. Heed the warning!

9.4 Variable parameters

Now that you know something about pointers, we can discuss variable parameters and passing by reference in more detail. (See Chapter 8 [Parameters], page 39, to refresh your memory on this topic.)

There are two main ways to return information from a function. The most common way uses the `return` command. However, `return` can only pass one value at a time back to the calling function. The second way to return information to a function uses variable parameters. Variable parameters ("passing by reference") enable you to pass back an arbitrary number of values, as in the following example:

```
#include <stdio.h>

int main();
void get_values (int *, int *);

int main()
{
  int num1, num2;
  get_values (&num1, &num2);
```

```
  printf ("num1 = %d and num2 = %d\n\n", num1, num2);

  return 0;
}


void get_values (int *num_ptr1, int *num_ptr2)
{
  *num_ptr1 = 10;
  *num_ptr2 = 20;
}
```

The output from this program reads:

```
num1 = 10 and num2 = 20
```

Note that we do use a `return` command in this example — in the `main` function. Remember, `main` must always be declared of type `int` and should always return an integer value. (See Chapter 22 [Style], page 219.)

When you use value parameters, the formal parameters (the parameters in the function being called) are mere copies of the actual parameters (the parameters in the function call). When you use variable parameters, on the other hand, you are passing the addresses of the variables themselves. Therefore, in the program above, it is not copies of the variables `num1` and `num2` that are passed to `get_values`, but the addresses of their actual memory locations. This information can be used to alter the variables directly, and to return the new values.

9.4.1 Passing pointers correctly

You might be wondering why `main` calls the function `get_values` above with ampersands before the parameters —

```
get_values (&num1, &num2);
```

— while the function itself is defined with asterisks before its parameters:

```
void get_values (int *num_ptr1, int *num_ptr2)
{
  *num_ptr1 = 10;
  *num_ptr2 = 20;
}
```

Think carefully for a moment about what is happening in these fragments of code. The variables `num1` and `num2` in `main` are ordinary integers, so when `main` prefixes them with ampersands (`&`) while passing them to `get_values`, it is really passing integer pointers. Remember, `&num1` should be read as "the address of the variable `num1`".

The code reads like this:

```
get_values (&num1, &num2);
```

"Evaluate the function `get_values`, passing to it the addresses at which the variables `num1` and `num2` are stored.".

The function `get_values` is defined like this:

```
void get_values (int *num_ptr1, int *num_ptr2)
```

> "Define the function `get_values`. It returns a `void` value (so it operates only via "side effects" on the variable parameters it is passed). It takes two parameters, both of type `int *`. The first parameter is called `num_ptr1` and is a pointer to an integer value, and the second parameter is called `num_ptr2` and is also a pointer to an integer value. When this function is called, it must be passed the addresses of variables, not the variables themselves."

Remember that declaring a variable with an asterisk (`*`) before it means "declare this variable to be a pointer", so the formal parameters of `get_values` are integer pointers. The parameters *must* be declared this way, because the `main` function sends the addresses of `num1` and `num2` — that is, by the time the `get_values` function receives the parameters, they are *already* pointers — hence their names in `get_values`: `num_ptr1` and `num_ptr2`, rather than `num1` and `num2`.

In effect, we are "matching up" the data types of `num1` and `num2` with those of `num_ptr1` and `num_ptr2`, respectively, when we prefix `num1` and `num2` with ampersands while passing them, and prefix `num_ptr1` and `num_ptr2` with asterisks in the parameter list of the function `get_values`. We do not have to write `num_ptr1 = &num1;` and `num_ptr2 = &num2;` — the calling convention does that for us.

Important! This is a general rule in C: when you pass actual parameters as pointers using ampersands (e.g. `&num1`, "the address of the variable `num1`"), you must use asterisks to declare as pointers the corresponding formal parameters in the function to which you pass them, (e.g. `int *num_ptr1`, "the contents of the location pointed to by `num_ptr1`").

9.4.2 Another variable parameter example

There is nothing mysterious about pointers, but they can be tricky. Here is another example.

Notice that the pointers in both this example and the example above are dereferenced with asterisks before they are used (for instance, when the contents of the location pointed to by `height_ptr` are multiplied by the integer `hscale` with the line `*height_ptr = *height_ptr * hscale;` in the function `scale_dimensions` below).

```
#include <stdio.h>

int main();
void scale_dimensions (int *, int *);

/* Scale some measurements */

int main()
{
  int height,width;

  height = 4;
  width = 5;
```

```
  scale_dimensions (&height, &width);

  printf ("Scaled height = %d\n", height);
  printf ("Scaled width = %d\n", width);

  return 0;
}


void scale_dimensions (int *height_ptr, int *width_ptr)
{
  int hscale = 3;        /* scale factors */
  int wscale = 5;

  *height_ptr = *height_ptr * hscale;
  *width_ptr = *width_ptr * wscale;
}
```

9.5 Questions for Chapter 9

1. What is a pointer?
2. How is a variable declared to be a pointer?
3. What data types can pointers point to?
4. Write a statement which converts a pointer to an integer into a pointer to a `double` type.
5. Why is it incorrect to write `float *number = 2.65;` ?

10 Decisions

Testing and Branching. Making conditions.

Until now, our code examples have been linear: control has flowed in one direction from start to finish. In this chapter, we will examine ways to enable code to make decisions and to choose among options. You will learn how to program code that will function in situations similar to the following:

- If the user hits the jackpot, print a message to say so: 'You've won!'
- If a bank balance is positive, then print 'C' for "credit"; otherwise, print 'D' for "debit".
- If the user has typed in one of five choices, then do something that corresponds to the choice, otherwise display an error message.

In the first case there is a simple "do or don't" choice. In the second case, there are two choices. The final case contains several possibilities.

C offers four main ways of coding decisions like the ones above. They are listed below.

`if...`

```
if (condition)
{
  do something
}
```

`if...else...`

```
if (condition)
{
  do something
}
else
{
  do something else
}
```

`...?...:...`

```
(condition) ? do something : do something else;
```

`switch`

```
switch (condition)
{
  case first case : do first thing
  case second case : do second thing
  case third case : do third thing
}
```

10.1 `if`

The first form of the `if` statement is an all-or-nothing choice: if some condition is satisfied, do something; otherwise, do nothing. For example:

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

or

```
if (condition)
{
  compound statement
}
```

In the second example, instead of a single statement, a whole block of statements is executed. In fact, wherever you can place a single statement in C, you can place a *compound statement* instead: a block of statements enclosed by curly brackets.

A *condition* is usually an expression that makes some sort of comparison. It must be either true or false, and it must be enclosed in parentheses: '`(...)`'. If the condition is true, then the statement or compound statement following the condition will be executed; otherwise, it will be ignored. For example:

```
if (my_num == 0)
{
  printf ("The number is zero.\n");
}

if (my_num > 0)
{
  printf ("The number is positive.\n");
}

if (my_num < 0)
{
  printf ("The number is negative.\n");
}
```

The same code could be written more compactly in the following way:

```
if (my_num == 0) printf ("The number is zero.\n");
if (my_num > 0) printf ("The number is positive.\n");
if (my_num < 0) printf ("The number is negative.\n");
```

It is often a good idea stylistically to use curly brackets in an `if` statement. It is no less efficient from the compiler's viewpoint, and sometimes you will want to include more statements later. It also makes `if` statements stand out clearly in the code. However, curly brackets make no sense for short statements such as the following:

```
if (my_num == 0) my_num++;
```

The `if` command by itself permits only limited decisions. With the addition of `else` in the next section, however, `if` becomes much more flexible.

10.2 `if... else...`

Let's review the basic form of the `if... else...` statement:

```
if (condition)
{
  compound statement
}
else
{
```

```
  compound statement
}
```

As with the bare `if` statement, there is a simplified version of the `if... else...` statement without code blocks:

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

When the `if... else...` is executed, the condition in parentheses is evaluated. If it is true, then the first statement or code block is executed; otherwise, the second statement or code block is executed. This can save unnecessary tests and make a program more efficient:

```
if (my_num > 0)
{
  printf ("The number is positive.");
}
else
{
  printf ("The number is zero or negative.");
}
```

It is not necessary to test `my_num` in the second block because that block is not executed unless `my_num` is *not* greater than zero.

10.3 Nested `if` statements

Consider the following two code examples. Their purposes are exactly the same.

```
int my_num = 3;

if ((my_num > 2) && (my_num < 4))
{
  printf ("my_num is three");
}
```

or:

```
int my_num =3;

if (my_num > 2)
{
  if (my_num < 4)
  {
    printf ("my_num is three");
  }
}
```

Both of these code examples have the same result, but they arrive at it in different ways. The first example, when translated into English, might read, "If `my_num` is greater than two and `my_num` is less than four (and `my_num` is an integer), then `my_num` has to be three." The second method is more complicated. In English, it can be read, "If `my_num` is greater than two, do what is in the first code block. Inside it, `my_num` is always greater than two; otherwise the program would never have arrived there. Now, if `my_num` is also less than four, then do what is inside the second code block. Inside that block, `my_num` is always less than four. We also know it is more than two, since the whole of the second test happens inside the block where that's true. So, assuming `my_num` is an integer, it must be three."

In short, there are two ways of making compound decisions in C. You make nested tests, or you can use the comparison operators `&&`, `||`, and so on. In situations where sequences of comparison operators become too complex, nested tests are often a more attractive option.

Consider the following example:

```
if (i > 2)
{
  /* i is greater than 2 here! */
}
else
{
  /* i is less than or equal to 2 here! */
}
```

The code blocks in this example provide "safe zones" wherein you can rest assured that certain conditions hold. This enables you to think and code in a structured way.

You can nest `if` statements in multiple levels, as in the following example:

```
#include <stdio.h>

int main ()
{
  int grade;

  printf("Type in your grade: ");
  scanf ("%d", &grade);

  if (grade < 10)
  {
    printf ("Man, you're lame!  Just go away.\n");
  }
  else
  {
    if (grade < 65)
    {
      printf ("You failed.\n");
    }
    else
    {
      printf ("You passed!\n");
      if (grade >= 90)
      {
        printf ("And you got an A!\n");
      }
      else
      {
        printf ("But you didn't get an A.  Sorry.\n");
      }
```

```
    }
  }
  return 0;
}
```

10.4 The ?...:... operator

The `?...:...` operator is a sort of shorthand `if...else...` statement. Because it is a little cryptic, it is not often used, but the basic form is as follows:

(*condition*) ? *expression1* : *expression2*;

The program evaluates *condition*. If it is true (not zero), then *expression1* is returned; otherwise, *expression2* is returned.

For example, in the short program below, the line `bas = (foo > bar) ? foo : bar;` assigns `foo` to `bas` if `foo` is greater than `bar`; otherwise, it assigns `bar` to `bas`.

```
#include <stdio.h>

int main()
{
  int foo = 10;
  int bar = 50;
  int bas;

  bas = (foo > bar) ? foo : bar;

  printf("bas = %d\n\n", bas);

  return 0;
}
```

The program will print '`bas = 50`' as a result.

10.5 The switch statement

The `switch` construction is another way of making decisions in C code. It is very flexible, but only tests for integer and character values. It has the following general form:

```
switch (integer or character expression)
{
  case constant1 : statement1;
  break;          /* optional */

  case constant2 : statement2;
  break;          /* optional */

  case constant3 : statement3;
  break;          /* optional */
  ...
}
```

The integer or character expression in the parentheses is evaluated, and the program checks whether it matches one of the constants in the various *cases* listed. If there is a match, the statement following that case will be executed, and execution will continue until either a `break` statement or the closing curly bracket of the entire `switch` statement is encountered.

One of the cases is called `default`. Statements after the `default` case are executed when none of the other cases are satisfied. You only need a default case if you are not sure you are covering every case with the ones you list.

Here is an example program that uses the `switch` statement to translate decimal digits into Morse code:

10.6 Example Listing

```
#include <stdio.h>

int main ();
void morse (int);


int main ()
{
  int digit;

  printf ("Enter any digit in the range 0 to 9: ");
  scanf ("%d", &digit);

  if ((digit < 0) || (digit > 9))
  {
    printf ("Your number was not in the range 0 to 9.\n");
  }
  else
  {
    printf ("The Morse code of that digit is ");
    morse (digit);
  }
  return 0;
}


void morse (int digit)        /* print out Morse code */
{
  switch (digit)
  {
    case 0 : printf ("-----");
      break;
    case 1 : printf (".----");
      break;
    case 2 : printf ("..---");
      break;
```

```
    case 3 : printf ("...--");
      break;
    case 4 : printf ("....-");
      break;
    case 5 : printf (".....");
      break;
    case 6 : printf ("-....");
      break;
    case 7 : printf ("--...");
      break;
    case 8 : printf ("---..");
      break;
    case 9 : printf ("----.");
  }
  printf ("\n\n");
}
```

The `morse` function selects one of the `printf` statements with `switch`, based on the integer expression `digit`. After every `case` in the switch, a `break` statement is used to jump `switch` statement's closing bracket '}'. Without `break`, execution would *fall through* to the next case and execute its `printf` statement.

Here is an example of using *fallthrough* in a constructive way. The function `yes` accepts input from the user and tests whether it was 'y' or 'Y'. (The `getchar` function is from the standard library and reads a character of input from the terminal. See Section 16.3.1 [getchar], page 138.)

```
#include <stdio.h>

int main ()
{
  printf ("Will you join the Free Software movement? ");
  if (yes())
  {
    printf("Great!  The price of freedom is eternal vigilance!\n\n");
  }
  else
  {
    printf("Too bad.  Maybe next life...\n\n");
  }

  return 0;
}


int yes()
{
  switch (getchar())
  {
    case 'y' :
    case 'Y' : return 1;
```

```
      default  : return 0;
    }
  }
```

If the character is '`y`', then the program falls through and meets the statement `return 1`. If there were a `break` statement after `case 'y'`, then the program would not be able to reach `case 'Y'` unless an actual '`Y`' were typed.

Note: The `return` statements substitute for `break` in the above code, but they do more than break out of `switch` — they break out of the whole function. This can be a useful trick.

10.7 Questions for Chapter 10

1. Translate the following into good C: "If 1 does not equal 42, print out 'Thank heavens for mathematics!' "
2. Write a program to get a lot of numbers from the user and print out the maximum and minimum of those.
3. Write an automatic teller machine program that simulates telling you your bank balance when you enter your account number and PIN number, but otherwise displays an error.
4. Write a mock program for a car computer that tells you how many kilometers to the liter you're getting when you enter how many liters of gas you've used and how far you travelled.

11 Loops

Controlling repetitive processes. Nesting loops

Loops are a kind of C construct that enable the programmer to execute a sequence of instructions over and over, with some condition specifying when they will stop. There are three kinds of loop in C:

- `while`
- `do ... while`
- `for`

11.1 `while`

The simplest of the three is the `while` loop. It looks like this:

```
while (condition)
{
  do something
}
```

The condition (for example, `(a > b)`) is evaluated every time the loop is executed. If the condition is true, then statements in the curly brackets are executed. If the condition is false, then those statements are ignored, and the `while` loop ends. The program then executes the next statement in the program.

The condition comes at the start of the loop, so it is tested at the start of every *pass*, or time through the loop. If the condition is false before the loop has been executed even once, then the statements inside the curly brackets will never be executed. (See Section 11.2 [do...while], page 62, for an example of a loop construction where this is not true.)

The following example prompts the user to type in a line of text, and then counts all the spaces in the line. The loop terminates when the user hits the ⟨RET⟩ key and then prints out the number of spaces. (See Section 16.3.1 [getchar], page 138, for more information on the standard library `getchar` function.)

```
#include <stdio.h>

int main()
{
  char ch;
  int count = 0;

  printf ("Type in a line of text.\n");

  while ((ch = getchar()) != '\n')
  {
    if (ch == ' ')
    {
      count++;
    }
  }
```

```
  printf ("Number of spaces = %d.\n\n", count);
  return 0;
}
```

11.2 do...while

The `do..while` loop has the form:

```
do
{
  do something
}
while (condition);
```

Notice that the condition is at the *end* of this loop. This means that a `do..while` loop will always be executed at least once, before the test is made to determine whether it should continue. This is the chief difference between `while` and `do...while`.

The following program accepts a line of input from the user. If the line contains a string of characters delimited with double quotation marks, such as '`"Hello!"`', the program prints the string, with quotation marks. For example, if the user types in the following string:

```
I walked into a red sandstone building. "Oof!" [Careful, Nick!]
```

...then the program will print the following string:

```
"Oof!"
```

If the line contains only one double quotation mark, then the program will display an error, and if it contains no double quotation marks, the program will print nothing.

Notice that the `do...while` loop in `main` waits to detect a linefeed character (`\n`), while the one in `get_substring` looks for a double quotation mark ('`"`'), but checks for a linefeed in the *loop body*, or main code block of the loop, so that it can exit the loop if the user entered a linefeed prematurely (before the second '`"`').

This is one of the more complex examples we have examined so far, and you might find it useful to *trace* the code, or follow through it step by step.

```
#include <stdio.h>

int main();
void get_substring();


int main()
{
  char ch;

  printf ("Enter a string with a quoted substring:\n\n");

  do
  {
    ch = getchar();
    if (ch == '"')
    {
```

```
      putchar(ch);
      get_substring();
    }
  }
  while (ch != '\n');

  return 0;
}


void get_substring()
{
  char ch;

  do
  {
    ch = getchar();
    putchar(ch);

    if (ch == '\n')
    {
      printf ("\nString was not closed ");
      printf ("before end of line.\n");
      break;
    }
  }
  while (ch != '"');

  printf ("\n\n");
}
```

11.3 `for`

The most complex loop in C is the `for` loop. The `for` construct, as it was developed in earlier computer languages such as BASIC and Pascal, was intended to behave in the following way:

> For all values of *variable* from *value1* to *value2*, in steps of `value3`, repeat the following sequence of commands...

The `for` loop in C is much more versatile than its counterpart in those earlier languages. The `for` loop looks like this in C:

```
for (initialization; condition; increment)
{
  do something;
}
```

In normal usage, these expressions have the following significance.

- *initialization*

  This is an expression that initializes the *control variable*, or the variable tested in the *condition* part of the `for` statement. (Sometimes this variable is called the loop's *index*.) The *initialization* part is only carried out once before the start of the loop. Example: `index = 1`.

- *condition*

  This is a conditional expression that is tested every time through the loop, just as in a `while` loop. It is evaluated at the *beginning* of every loop, and the loop is only executed if the expression is true. Example: `index <= 20`.

- *increment*

  This is an expression that is used to alter the value of the control variable. In earlier languages, this usually meant adding or subtracting 1 from the variable. In C, it can be almost anything. Examples: `index++`, `index *= 20`, or `index /= 2.3`.

For example, the following `for` loop prints out the integers from 1 to 10:

```
int my_int;

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

The following example prints out all prime numbers between 1 and the macro value `MAX_INT`. (A prime numbers is a number that cannot be divided by any number except 1 and itself without leaving a remainder.) This program checks whether a number is a prime by dividing it by all smaller integers up to half its size. (See Chapter 12 [Preprocessor directives], page 71, for more information on macros.)

```
#include <stdio.h>

#define MAX_INT  500
#define TRUE      1
#define FALSE     0

int main ()
{
  int poss_prime;

  for (poss_prime = 2; poss_prime <= MAX_INT; poss_prime++)
  {
    if (prime(poss_prime))
    {
      printf ("%d ", poss_prime);
    }
  }
  printf("\n\n");
  return 0;
}


prime (int poss_prime)       /* check whether poss_prime is prime */
```

```
{
  int poss_factor;

  for (poss_factor = 2; poss_factor <= poss_prime/2; poss_factor++)
  {
    if (poss_prime % poss_factor == 0)
    {
      return (FALSE);
    }
  }

  return (TRUE);
}
```

The program should print the following sequence of integers:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
```

11.4 The flexibility of `for`

As mentioned above, C's `for` construct is quite versatile. You can use almost any statement you like for its *initialization*, *condition*, and *increment* parts, including an empty statement. For example, omitting the *initialization* and *increment* parts creates what is essentially a `while` loop:

```
int my_int = 1;

for ( ; my_int <= 20; )
{
  printf ("%d ", my_int);
  my_int++;
}
```

Omitting the *condition* part as well produces an *infinite loop*, or loop that never ends:

```
for ( ; ; )
{
  printf("Aleph Null bottles of beer on the wall...\n");
}
```

You can break out of an "infinite loop" with the `break` or `return` commands. (See Section 11.5 [Terminating and speeding loops], page 67.)

Consider the following loop:

```
for (my_int = 2; my_int <= 1000; my_int = my_int * my_int)
{
  printf ("%d ", my_int);
}
```

This loop begins with 2, and each time through the loop, `my_int` is squared.

Here's another odd `for` loop:

```
char ch;

for (ch = '*'; ch != '\n'; ch = getchar())
{
  /* do something */
}
```

This loop starts off by initializing `ch` with an asterisk. It checks that `ch` is not a linefeed character (which it isn't, the first time through), then reads a new value of `ch` with the library function `getchar` and executes the code inside the curly brackets. When it detects a line feed, the loop ends.

It is also possible to combine several *increment* parts in a `for` loop using the comma operator `,`. (See Section 18.2 [The comma operator], page 181, for more information.)

```
#include <stdio.h>

int main()
{
  int up, down;

  for (up = 0, down=10; up < down; up++, down--)
  {
    printf("up = %d, down= %d\n",up,down);
  }

  return 0;
}
```

The example above will produce the following output:

```
up = 0, down= 10
up = 1, down= 9
up = 2, down= 8
up = 3, down= 7
up = 4, down= 6
```

One feature of the `for` loop that unnerves some programmers is that even the value of the loop's conditional expression can be altered from within the loop itself:

```
int index, number = 20;

for (index = 0; index <= number; index++)
{
  if (index == 9)
  {
    number = 30;
  }
}
```

In many languages, this technique is syntactically forbidden. Not so in the flexible language C. It is rarely a good idea, however, because it can make your code confusing and hard to maintain.

11.5 Terminating and speeding loops

C provides simple ways of terminating or speeding up any of the three loops we have discussed, whether or not it has run its course. The three main commands to do so are `break`, `return`, and `continue`.

11.5.1 Terminating loops with break

The usual statement to terminate a loop is the same statement that is used to jump out of `switch` statements:

```
break;
```

If this statement is encountered within a loop, the loop will end immediately. For instance, here is an inefficient way of assigning 12 to `my_int`:

```
for (my_int = 1; my_int <= 100; my_int++)
{
  if (my_int == 12)
  {
    break;
  }
}

printf("my_int = %d\n\n", my_int);
```

11.5.2 Terminating loops with return

Suppose that a program is in the middle of a loop (or some nested loops) in a complex function, and suddenly the function finds its answer. This is where the `return` statement comes in handy. The `return` command will jump out of any number of loops and pass the value back to the calling function without having to finish the loops or the rest of the function. (See Section 11.6 [Nested loops], page 68, for clarification of the idea of placing one loop inside another.)

Example:

```
#include <stdio.h>

int main()
{
  printf ("%d\n\n", returner(5, 10));
  printf ("%d\n\n", returner(5, 5000));
  return 0;
}


int returner (int foo, int bar)
{
  while (foo <= 1000)
  {
    if (foo > bar)
    {
```

```
        return (foo);
      }

      foo++;
    }

    return foo;
  }
```

The function `returner` contains a `while` loop that increments the variable `foo` and tests it against a value of 1000. However, if at any point the value of `foo` exceeds the value of the variable `bar`, the function will exit the loop, immediately returning the value of `foo` to the calling function. Otherwise, when `foo` reaches 1000, the function will increment `foo` one more time and return it to `main`.

Because of the values it passes to `returner`, the `main` function will first print a value of 11, then 1001. Can you see why?

11.5.3 Speeding loops with `continue`

Instead of terminating a loop, you might want to speed it to its next pass, perhaps to avoid executing irrelevant statements. To do so, you should use the `continue` statement. When a `continue` statement is encountered, the program will skip the rest of the loop's code block and jump straight to the start of the next pass through the loop.

Here is an example that uses the `continue` statement to avoid division by zero (which causes a run-time error):

```
for (my_int = -10; my_int <= 10; my_int++)
{
  if (my_int == 0)
  {
    continue;
  }

  printf ("%d", 20/i);
}
```

11.6 Nested loops

Just as decisions can be nested, so can loops; that is, you can place loops inside other loops. This can be useful, for example, when you are coding multidimensional arrays. (See Chapter 14 [Arrays], page 89.)

The example below prints a square of asterisks by nesting a `printf` command inside an *inner loop*, which is itself nested inside an *outer loop*.

Any kind of loop can be nested. For example, the code below could have been written with `while` loops instead of `for` loops:

```
#include <stdio.h>

#define SIZE 5
```

```
int main()
{
  int square_y, square_x;

  printf ("\n");

  for (square_y = 1; square_y <= SIZE; square_y++)
  {
    for (square_x = 1; square_x <= SIZE; square_x++)
    {
      printf("*");
    }
    printf ("\n");
  }

  printf ("\n");
  return 0;
}
```

The output of the above code looks like this:

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

11.7 Questions for Chapter 11

1. How many kinds of loop does C offer, and what are they?
2. When is the condition tested in each of the loops?
3. Which of the loops is always executed at least once?
4. Write a program that copies all input to output line by line.
5. Write a program to get 10 numbers from the user and add them together.
6. Rewrite the nested loops example to print a square with `while` loops instead of `for` loops.

12 Preprocessor directives

Making programming versatile.

GCC, the GNU Compiler Collection, contains a C *preprocessor*. A preprocessor is a program that examines C code before it is compiled and manipulates it in various ways. There are two main uses of a preprocessor. One is to include external files, such as header files. The other is to define *macros*, which are names (possibly with arguments) that are expanded by the preprocessor into pieces of text or C code. Macros that are expanded into text are usually displayed to the user, but macros that are expanded into C code are executed with the rest of the C code that surrounds them.

12.1 A few directives

All preprocessor `directives`, or commands, are preceded by a hash mark ('`#`'). One example you have already seen in previous chapters is the `#include` directive:

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

This directive tells the preprocessor to *include* the file '`stdio.h`'; in other words, to treat it as though it were part of the program text.

A file to be included may itself contain `#include` directives, thus encompassing other files. When this happens, the included files are said to be *nested*.

Here are a few other directives:

`#if` ... `#endif`
: The `#if` directive is followed by an expression on the same line. The lines of code between `#if` and `#endif` will be compiled only if the expression is true. This is called *conditional compilation*.

`#else`
: This is part of an `#if` preprocessor statement and works in the same way with `#if` that the regular C `else` does with the regular `if`.

`#line` *constant filename*
: This causes the compiler to act as though the next line is line number *constant* and is part of the file *filename*. Mainly used for debugging.

`#error`
: This forces the compiler to abort. Also intended for debugging.

Below is an example of conditional compilation. The following code displays '`23`' to the screen.

```
#include <stdio.h>

#define CHOICE 500

int my_int = 0;

#if (CHOICE == 500)
void set_my_int()
{
  my_int = 23;
}
#else
void set_my_int()
{
  my_int = 17;
}
#endif

int main ()
{
  set_my_int();
  printf("%d\n", my_int);

  return 0;
}
```

12.2 Macros

Macros can make long, ungainly pieces of code into short words. The simplest use of macros is to give constant values meaningful names. For example:

```
#define MY_PHONE 5551234
```

This allows the programmer to use the word `MY_PHONE` to mean the number 5551234. In this case, the word is longer than the number, but it is more meaningful and makes a program read more naturally. It can also be centralised in a header file, where it is easily changed; this eliminates tedious search-and-replace procedures on code if the value appears frequently in the code. It has been said, with some humorous exaggeration, that the only values that should appear "naked" in C code instead of as macros or variables are 1 and 0.

The difference between defining a macro for 5551234 called `MY_PHONE` and declaring a long integer variable called `my_phone` with the same value is that the variable `my_phone` has the value 5551234 only provisionally; it can be incremented with the statement `my_phone++;`, for example. In some sense, however, the macro `MY_PHONE` *is* that value, and *only* that value — the C preprocessor simply searches through the C code before it is compiled and replaces every instance of `MY_PHONE` with `5551234`. Issuing the command `MY_PHONE++;` is no more or less sensible than issuing the command `5551234++;`.

Any piece of C code can be made into a macro, Macros are not merely constants referred to at compile time, but are strings that are physically replaced with their values by the preprocessor before compilation. For example:

```
#define SUM 1 + 2 + 3 + 4
```

would allow `SUM` to be used instead of `1 + 2 + 3 + 4`. Usually, this would equal 10, so that in the statement `example1 = SUM + 10;`, the variable `example1` equals 20. Sometimes, though, this macro will be evaluated differently; for instance, in the statement `example2 = SUM * 10;`, the variable `example2` equals 46, instead of 100, as you might think. Can you figure out why? Hint: it has to do with the order of operations.

The quotation marks in the following macro allow the string to be called by the identifier `SONG` instead of typing it out over and over. Because the text '`99 bottles of beer on the wall...`' is enclosed by double quotation marks, it will never be interpreted as C code.

```
#define SONG "99 bottles of beer on the wall..."
```

Macros cannot define more than a single line, but they can be used anywhere except inside strings. (Anything enclosed in string quotes is assumed to be untouchable by the compiler.)

Some macros are defined already in the file '`stdio.h`', for example, `NULL` (the value 0).

There are a few more directives for macro definition besides `#define`:

`#undef` This undefines a macro, leaving the name free.

`#ifdef` This is a kind of `#if` that is followed by a macro name. If that macro is defined then this directive is true. `#ifdef` works with `#else` in the same way that `#if` does.

`#ifndef` This is the opposite of `#ifdef`. It is also followed by a macro name. If that name is not defined then this is true. It also works with `#else`.

Here is a code example using some macro definition directives from this section, and some conditional compilation directives from the last section as well.

```
#include <stdio.h>

#define CHOICE 500

int my_int = 0;

#undef CHOICE
#ifdef CHOICE
void set_my_int()
{
  my_int = 23;
}
#else
void set_my_int()
{
  my_int = 17;
}
#endif

int main ()
{
  set_my_int();
  printf("%d\n", my_int);

  return 0;
}
```

The above code example displays '17' on the screen.

12.2.1 Macro functions

Macros can also accept parameters and return values. Macros that do so are called *macro functions*. To create a macro function, simply define a macro with a parameter that has whatever name you like, such as `my_val`. For example, one macro defined in the standard libraries is `abs`, which returns the absolute value of its parameter. Let us define our own version, `ABS`, below. (Note that we are defining it in upper case not only to avoid conflicting with `abs`, but also because all macros should be defined in upper case, in the GNU coding style. See Chapter 22 [Style], page 219.)

```
#define ABS(my_val) ((my_val) < 0) ? -(my_val) : (my_val)
```

This macro uses the `?...:...` command to return a positive number no matter what value is assigned to `my_val` — if `my_val` is defined as a positive number, the macro returns the same number, and if `my_val` is defined as a negative number, the macro returns its negative (which will be positive). (See Chapter 10 [Decisions], page 53, for more information on the `?...:...` structure. If you write `ABS(-4)`, then the preprocessor will substitute `-4` for `my_val`; if you write `ABS(i)`, then the preprocessor will substitute `i` for `my_val`, and so on. Macros can take more than one parameter, as in the code example below.

One caveat: macros are substituted whole wherever they are used in a program: this is potentially a huge amount of code repetition. The advantage of a macro over an actual function, however, is speed. No time is taken up in passing control to a new function, because control never leaves the home function; the macro just makes the function a bit longer.

A second caveat: function calls cannot be used as macro parameters. The following code will not work:

```
ABS (cos(36))
```

Here is an example of macro functions in use:

```
#include <stdio.h>

#define STRING1          "A macro definition\n"
#define STRING2          "must be all on one line!\n"
#define EXPRESSION1      1 + 2 + 3 + 4
#define EXPRESSION2      EXPRESSION1 + 10
#define ABS(x)           ((x) < 0) ? -(x) : (x)
#define MAX(a,b)         (a < b) ?  (b) : (a)
#define BIGGEST(a,b,c)   (MAX(a,b) < c) ?  (c) : (MAX(a,b))

int main ()
{
  printf (STRING1);
  printf (STRING2);
  printf ("%d\n", EXPRESSION1);
  printf ("%d\n", EXPRESSION2);
  printf ("%d\n", ABS(-5));
  printf ("Biggest of 1, 2, and 3 is %d\n", BIGGEST(1,2,3));

  return 0;
}
```

The output from the code example above is as follows:

```
A macro definition
must be all on one line!
10
20
5
Biggest of 1, 2, and 3 is 3
```

12.3 Extended macro example

Here are some examples of macros taken from actual working C code, in this case the code of GNU Emacs, the text editor of choice for many C programmers, and in fact the editor in which this edition of the book was written.

Most of the macro examples below define various types of integer as having certain sizes. It can be very useful when doing advanced C programming to know whether a long integer, for instance, is 32 or 64 bits long on your system; if you select the wrong size, your code might crash or might not even compile. In the case of Emacs, the maximum size of certain variables (how many bits they contain) affects every aspect of its operation, even determining how long an Emacs text file can be.

Each piece of code below is prefixed with the name of the file from which the code is taken, and followed by a note on some interesting features of the macros defined.

'emacs/src/config.h'

```
/* Note that lisp.h uses this in a preprocessor conditional, so it
   would not work to use sizeof.  That being so, we do all of them
   without sizeof, for uniformity's sake.  */
#ifndef BITS_PER_INT
#define BITS_PER_INT 32
#endif

#ifndef BITS_PER_LONG
#ifdef _LP64
#define BITS_PER_LONG 64
#else
#define BITS_PER_LONG 32
#endif
#endif
```

In the middle of this set of macros, from 'config.h', the Emacs programmer used the characters '/*' and '*/' to create an ordinary C comment. C comments can be interspersed with macros freely.

The macro `BITS_PER_INT` is defined here to be 32 (but only if it is not already defined, thanks to the `#ifndef` directive). The Emacs code will then treat integers as having 32 bits. (See Section 5.1 [Integer variables], page 19.)

The second chunk of macro code in this example checks to see whether `BITS_PER_LONG` is defined. If it is not, but `_LP64` is defined, it defines `BITS_PER_LONG` to be 64, so that all long integers will be treated as having 64 bits. (`_LP64` is a GCC macro that is defined on 64-bit systems. It stands for "longs and pointers are 64 bits".) If `_LP64` is *not* present, the code assumes it is on a 32-bit system and defines `BITS_PER_LONG` to be 32.

'emacs/src/lisp.h'

```
/* These are default choices for the types to use.  */
#ifdef _LP64
#ifndef EMACS_INT
#define EMACS_INT long
#define BITS_PER_EMACS_INT BITS_PER_LONG
#endif
#ifndef EMACS_UINT
#define EMACS_UINT unsigned long
#endif
#else /* not _LP64 */
#ifndef EMACS_INT
#define EMACS_INT int
#define BITS_PER_EMACS_INT BITS_PER_INT
#endif
#ifndef EMACS_UINT
#define EMACS_UINT unsigned int
#endif
#endif
```

This set of macros, from 'lisp.h', again checks to see whether _LP64 is defined. If it is, it defines EMACS_INT as long (if it is not already defined), and BITS_PER_EMACS_INT to be the same as BITS_PER_LONG, which was defined in 'config.h', above. It then defines EMACS_UINT to be an unsigned long, if it is not already defined.

If _LP64 is *not* defined, it is assumed we are on a 32-bit system. EMACS_INT is defined to be an int if it is not already defined, and EMACS_UINT is defined to be an unsigned int if it is not already defined.

Again, note that the programmer has freely interspersed a comment with the preprocessor code.

'emacs/src/lisp.h'

```
/* These values are overridden by the m- file on some machines.  */
#ifndef VALBITS
#define VALBITS (BITS_PER_EMACS_INT - 4)
#endif
```

Here is another example from 'lisp.h'. The macro VALBITS, which defines another size of integer internal to Emacs, is defined as four less than BITS_PER_EMACS_INT — that is, 60 on 64-bit systems, and 28 on 32-bit systems.

'emacs/src/lisp.h'

```
#ifndef XINT   /* Some machines need to do this differently.  */
#define XINT(a) ((EMACS_INT) (((a) << (BITS_PER_EMACS_INT - VALBITS)) \
                                 >> (BITS_PER_EMACS_INT - VALBITS)))
#endif
```

The interesting feature of the XINT macro above is not only that it is a function, but that it is broken across multiple lines with the backslash character ('\'). The GCC preprocessor simply deletes the backslash, deletes the preceding whitespace from the next line, and appends it where the backslash was. In this way, it is possible to treat long, multi-line macros as though they are actually on a single line. (See Chapter 18 [Advanced operators], page 177, for more information on the the advanced operators << and >>.)

12.4 Questions

1. Define a macro called `BIRTHDAY` which equals the day of the month upon which your birthday falls.
2. Write an instruction to the preprocessor to include the math library '`math.h`'.
3. A macro is always a number. True or false?

13 Libraries

Plug-in C expansions. Header files.

The core of the C language is small and simple, but special functionality is provided in the form of *external libraries* of ready-made functions. Standardized libraries make C code extremely *portable*, or easy to compile on many different computers.

Libraries are files of ready-compiled code that the compiler merges, or *links*, with a C program during compilation. For example, there are libraries of mathematical functions, string handling functions, and input/output functions. Indeed, most of the facilities C offers are provided as libraries.

Some libraries are provided for you. You can also make your own, but to do so, you will need to know how GNU builds libraries. We will discuss that later. (See Section 17.6 [Building a library], page 172.)

Most C programs include at least one library. You need to ensure both that the library is linked to your program and that its header files are included in your program.

The standard C library, or '`glibc`', is linked automatically with every program, but header files are never included automatically, so you must always include them yourself. Thus, you must always include '`stdio.h`' in your program if you intend to use the standard input/output features of C, even though '`glibc`', which contains the input/output routines, is linked automatically.

Other libraries, however, are not linked automatically. You must link them to your program yourself. For example, to link the math library '`libm.so`', type

```
gcc -o program_name program_name.c -lm
```

The command-line option to link '`libm.so`' is simply '`-lm`', without the '`lib`' or the '`.so`', or in the case of static libraries, '`.a`'. (See Section 13.2 [Kinds of library], page 81.)

The '`-l`' option was created because the average GNU system already has many libraries, and more can be added at any time. This means that sometimes two libraries provide alternate definitions of the same function. With judicious use of the '`-l`' option, however, you can usually clarify to the compiler which definition of the function should be used. Libraries specified earlier on the command line take precedence over those defined later, and code from later libraries is only linked in if it matches a *reference* (function definition, macro, global variable, etc.) that is still undefined. (See Section 17.4 [Compiling multiple files], page 166, for more information.)

In summary, you must always do two things:

- link the library with a '`-l`' option to gcc (a step that may be skipped in the case of '`glibc`').
- include the library header files (a step you must always follow, even for '`glibc`').

13.1 Header files

As mentioned above, libraries have *header files* that define information to be used in conjunction with the libraries, such as functions and data types. When you include a header file, the compiler adds the functions, data types, and other information in the header file to the list of reserved words and commands in the language. After that, you cannot use the names of functions or macros in the header file to mean anything other than what the library specifies, in any source code file that includes the header file.

The most commonly used header file is for the standard input/output routines in '`glibc`' and is called '`stdio.h`'. This and other header files are included with the `#include` command at the top of a source code file. For example,

```
#include "name.h"
```

includes a header file from the current directory (the directory in which your C source code file appears), and

```
#include <name.h>
```

includes a file from a *system directory* — a standard GNU directory like '`/usr/include`'. (The `#include` command is actually a *preprocessor directive*, or instruction to a program used by the C compiler to simplify C code. (See Chapter 12 [Preprocessor directives], page 71, for more information.)

Here is an example that uses the `#include` directive to include the standard '`stdio.h`' header in order to print a greeting on the screen with the `printf` command. (The characters '`\n`' cause `printf` to move the cursor to the next line.)

```
#include <stdio.h>

int main ()
{
  printf ("C standard I/O file is included.\n");
  printf ("Hello world!\n");

  return 0;
}
```

If you save this code in a file called '`hello.c`', you can compile this program with the following command:

```
gcc -o hello hello.c
```

As mentioned earlier, you can use some library functions without having to link library files explicitly, since every program is always linked with the *standard C library*. This is called '`libc`' on older operating systems such as Unix, but '`glibc`' ("GNU libc") on GNU systems. The '`glibc`' file includes standard functions for input/output, date and time calculation, string manipulation, memory allocation, mathematics, and other language features.

Most of the standard '`glibc`' functions can be incorporated into your program just by using the `#include` directive to include the proper header files. For example, since '`glibc`' includes the standard input/output routines, all you need to do to be able to call `printf` is put the line `#include <stdio.h>` at the beginning of your program, as in the example that follows.

Note that '`stdio.h`' is just one of the many header files you will eventually use to access '`glibc`'. The GNU C library is automatically linked with every C program, but you will eventually need a variety of header files to access it. These header files are not included in your code automatically — you must include them yourself!

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

int main ()
{
  double x, y;

  y = sin (x);
  printf ("Math library ready\n");

  return 0;
}
```

However, programs that use a special function outside of '`glibc`' — including mathematical functions that are nominally part of '`glibc`', such as function `sin` in the example above! — must use the '`-l`' option to `gcc` in order to link the appropriate libraries. If you saved this code above in a file called '`math.c`', you could compile it with the following command:

```
gcc -o math math.c -lm
```

The option '`-lm`' links in the library '`libm.so`', which is where the mathematics routines are actually located on a GNU system.

To learn which header files you must include in your program in order to use the features of '`glibc`' that interest you, consult section "Table of Contents" in *The GNU C Library Reference Manual*. This document lists all the functions, data types, and so on contained in '`glibc`', arranged by topic and header file. (See Section 13.3 [Common library functions], page 82, for a partial list of these header files.)

Note: Strictly speaking, you need not always use a system header file to access the functions in a library. It is possible to write your own declarations that mimic the ones in the standard header files. You might want to do this if the standard header files are too large, for example. In practice, however, this rarely happens, and this technique is better left to advanced C programmers; using the header files that came with your GNU system is a more reliable way to access libraries.

13.2 Kinds of library

There are two kinds of library: *static libraries* and *shared libraries*. When you link to a static library, the code for the entire library is merged with the object code for your program. If you link to many static libraries, your executable will be enormous.

Shared libraries were developed in the late 1980s to reduce the code size of programs on operating systems like GNU. When you link to a shared library, the library's code is not merged with your program's object code. Instead, *stub code* is inserted into your object code. The stub code is very small and merely calls the functions in the shared library — the operating system does the rest. An executable created with a shared library can therefore be far smaller than one created with a static library. Shared libraries can also reduce the amount of memory used.

Although shared libraries seem to have every advantage over static libraries, static libraries are still useful. For example, sometimes you will wish to distribute an executable to people whose computers do not have the libraries that yours does. In that case, you might link to a static version of the libraries. This will incorporate the library functions that you need into your executable, so that it will run on systems that don't have those libraries. (It is also sometimes easier to debug a program that is linked to static libraries than one linked to shared libraries. See Section 23.5 [Introduction to GDB], page 230, for more information.)

The file name for a library always starts with '`lib`' and ends with either '`.a`' (if it is static) or '`.so`' (if it is shared). For example, '`libm.a`' is the static version of the C math library, and '`libm.so`' is the shared version. As explained above, you must use the '`-l`' option with the name of a library, minus its '`lib`' prefix and '`.a`' or '`.so`' suffix, to link that library to your program (except the library '`glibc`', which is always linked). For example, the following shell command creates an executable program called '`math`' from the source code file '`math.c`' and the library '`libm.so`'.

```
gcc -o math math.c -lm
```

The shared version of the library is always linked by default. If you want to link the static version of the library, you must use the GCC option '`--static`'. The following example links '`libm.a`' instead of '`libm.so`'.

```
gcc -o math math.c -lm --static
```

Type '`info gcc`' at your shell prompt for more information about GCC options.

13.3 Common library functions

Checking character types. Handling strings. Doing maths.

The libraries in GCC contain a repertoire of standard functions and macros. There are many different kinds of function and macro in the libraries. Here are a few of the different kinds available, with the header files you can use to access them:

- Character handling: '`ctype.h`'
- Mathematics: '`math.h`'
- String manipulation: '`string.h`'

You may find it useful to read the header files yourself. They are usually found in the directories '`/usr/include`' and its subdirectories on GNU/Linux systems. The three header files listed above can be found in '`/usr/include`'; there is a second version of '`ctype.h`' in '`/usr/include/linux`'.[1]

13.3.1 Character handling

Let's examine some simple library functions and see how they are used. Some of the functions that are available on GNU systems for handling individual characters are described below. They are all macros, so the usual caveats about macro parameters apply. (See

[1] The version of '`ctype.h`' in the '`/usr/include`' directory proper is the one that comes with '`glibc`'; the one in '`/usr/include/linux`' is a special version associated with the Linux kernel. You can specify the one you want with a full pathname inside double quotes (for example, `#include "/usr/include/linux/ctype.h"`), or you can use the '`-I`' option of `gcc` to force GCC to search a set of directories in a specific order. See Section 17.6 [Building a library], page 172, for more information.)

Section 12.2.1 [Macro functions], page 74.) All of the functions below accept single variables of type `char` as parameters. To use any of them, you must include the system header file '`ctype.h`'; the library used is simply `glibc`, which is linked automatically.

`isalnum` Returns true if and only if the parameter is alphanumeric: that is, an alphabetic character (see `isalpha`) or a digit (see `isdigit`).

`isalpha` Returns true if and only if the parameter is alphabetic. An *alphabetic* character is any character from '`A`' through '`Z`' or '`a`' through '`z`'.

`isascii` Returns true if and only if the parameter is a valid ASCII character: that is, it has an integer value in the range 0 through 127. (Remember, the `char` type in C is actually a kind of integer!)

`iscntrl` Returns true if and only if the parameter is a control character. Control characters vary from system to system, but are usually defined as characters in the range 0 to 31.

`isdigit` Returns true if and only if the parameter is a digit in the range 0 through 9.

`isgraph` Returns true if and only if the parameter is graphic: that is, if the character is either alphanumeric (see `isalnum`) or punctuation (see `ispunct`). All graphical characters are valid ASCII characters, but ASCII also includes non-graphical characters such as control characters (see `iscntrl`) and whitespace (see `isspace`).

`islower` Returns true if and only if the parameter is a lower-case alphabetic character (see `isalpha`).

`isprint` Returns true if and only if the parameter is a printable character: that is, the character is either graphical (see `isgraph`) or a space character.

`ispunct` Returns true if and only if the parameter is a punctuation character.

`isspace` Returns true if and only if the parameter is a whitespace character. What is defined as whitespace varies from system to system, but it usually includes space characters and tab characters, and sometimes newline characters.

`isupper` Returns true if and only if the parameter is an upper-case alphabetic character (see `isalpha`).

`isxdigit` Returns true if and only if the parameter is a valid hexadecimal digit: that is, a decimal digit (see `isdigit`), or a letter from '`a`' through '`f`' or '`A`' through '`F`'.

`toascii` Returns the parameter stripped of its eighth bit, so that it has an integer value from 0 through 127 and is therefore a valid ASCII character. (See `isascii`.)

`tolower` Converts a character into its lower-case counterpart. Does not affect characters which are already in lower case.

`toupper` Converts a character into its upper-case counterpart. Does not affect characters which are already in upper case.

```
/********************************************************/
/*                                                      */
/* Demonstration of character utility functions         */
/*                                                      */
/********************************************************/

#include <stdio.h>
#include <ctype.h>

#define  allchars    ch = 0; isascii(ch); ch++

int main ()           /* A criminally long main program! */

{
  char ch;

  printf ("\n\nVALID CHARACTERS FROM isgraph:\n\n");
  for (allchars)
    {
      if (isgraph(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM isalnum:\n\n");
  for (allchars)
    {
      if (isalnum(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM isalpha:\n\n");
  for (allchars)
    {
      if (isalpha(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM isupper:\n\n");
  for (allchars)
    {
      if (isupper(ch))
{
  printf ("%c ",ch);
```

```
}
    }

  printf ("\n\nVALID CHARACTERS FROM islower:\n\n");
  for (allchars)
    {
      if (islower(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM isdigit:\n\n");
  for (allchars)
    {
      if (isdigit(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM isxdigit:\n\n");
  for (allchars)
    {
      if (isxdigit(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\nVALID CHARACTERS FROM ispunct:\n\n");
  for (allchars)
    {
      if (ispunct(ch))
{
  printf ("%c ",ch);
}
    }

  printf ("\n\n");

  return 0;
}
```

The output of the above code example is as follows:

```
VALID CHARACTERS FROM isgraph:

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h
i j k l m n o p q r s t u v w x y z { | } ~
```

```
VALID CHARACTERS FROM isalnum:

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM isalpha:

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

VALID CHARACTERS FROM isupper:

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

VALID CHARACTERS FROM islower:

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

VALID CHARACTERS FROM isdigit:

0 1 2 3 4 5 6 7 8 9

VALID CHARACTERS FROM isxdigit:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

VALID CHARACTERS FROM ispunct:

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

13.4 Mathematical functions

Let us now examine some simple math library functions. (This section presupposes some familiarity on your part with trigonometry. If you have none, you might want to skip this section for now – but reading it won't hurt you!)

The following mathematical functions, among others, are available on GNU systems. Many of the functions described below are macros, so the usual caveats about macro parameters apply. (See Section 12.2.1 [Macro functions], page 74.) All of these functions require parameters of type `double` or `long float`. Constants used must be written in floating point form: for instance, write '`7.0`' instead of just '`7`'.

Here is a list of some functions you can expect to find in the headers '`math.h`', '`tgmath.h`', and '`limits.h`'.

`abs` Returns the unsigned value of the parameter in brackets. This function is a macro; see `fabs` for a proper function version.

`acos` Returns the arccosine (or inverse cosine) of the parameter, which must lie between -1.0 and +1.0 inclusive. (The result is always in radians.)

`asin` Returns the arcsine (or inverse sine) of the parameter, which must lie between -1.0 and +1.0 inclusive. (The result is always in radians.)

`atan` Returns the arctangent (or inverse tangent) of the parameter. (The result is always in radians.)

`atan2` This is a special function for calculating the inverse tangent of the second parameter divided by the first. `atan2` will find the result more accurately than `atan` will.

```
result = atan2 (x, y);
result = atan2 (x, 3.14);
```

`ceil` Returns the ceiling for the parameter: that is, the integer just above it. In effect, rounds the parameter up.

`cos` Returns the cosine of the parameter in radians. (The parameter is also assumed to be specified in radians.)

`cosh` Returns the hyperbolic cosine of the parameter.

`exp` Returns the exponential function of the parameter (i.e. e to the power of the parameter).

`fabs` Returns the absolute or unsigned value of the parameter in brackets. This is the version that is a proper function; see `abs` if you want one that is a macro.

`floor` Returns the floor for the parameter: that is, the integer just below it. In effect, rounds the parameter down to the nearest integer value, i.e. truncates it.

`log` Returns the natural logarithm of the parameter. The parameter used must be greater than zero, but does not have to be declared as unsigned.

`log10` Returns the base 10 logarithm of the parameter. The parameter used must be greater than zero, but does not have to be declared as unsigned.

`pow` Returns the first parameter raised to the power of the second.

```
result = pow (x,y); /*raise x to the power y */
result = pow (x,2); /* square x */
```

`sin` Returns the sine of the parameter in radians. (The parameter is also assumed to be specified in radians.)

`sinh` Returns the hyperbolic sine of the parameter. (Pronounced “shine” or “sinch”.)

`sqrt` Returns the positive square root of the parameter.

`tan` Returns the tangent of the parameter in radians. (The parameter is also assumed to be specified in radians.)

`tanh` Returns the hyperbolic tangent of the parameter.

Here is a code example that uses a few of the math library routines listed above.

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

int main()
{
  double my_pi;

  my_pi = 4 * atan(1.0);

  /* Print the value of pi we just calculated, to 32 digits. */
  printf ("my_pi = %.32f\n", my_pi);

  /* Print value of pi from math library, to 32 digits. */
  printf ("M_PI  = %.32f\n", M_PI);

  return 0;
}
```

If you save the above example as 'pi.c', you will have to enter a command such as the one below to compile it.

```
gcc pi.c -o pi -lm
```

When you compile and run the code example, it should print the following results:

```
my_pi = 3.14159265358979311599796346854419
M_PI  = 3.14159265358979311599796346854419
```

13.5 Questions for Chapter 13

1. How do you incorporate a library file into a C program?
2. Name the most commonly used library file in C.
3. Is it possible to define new functions with the same names as standard library functions?
4. What type of data is returned from mathematical functions?
5. True or false? All mathematical calculations are performed using doubles.
6. Name five kinds of error which can occur in a mathematical function.

14 Arrays

Rows and tables of storage.

Suppose you have a long list of numbers, but you don't want to assign them to variables individually. For example, you are writing a simple program for a restaurant to keep a list of the amount each diner has on his or her tab, but you don't want to go through the tedium of writing a list like the following:

```
float alfies_tab, bettys_tab, charlies_tab ...;

alfies_tab = 88.33;
bettys_tab = 17.23;
charlies_tab = 55.55;
etc.
```

A list like that could run to hundreds or thousands of entries, and for each diner you'd have to write "special-case" code referring to every diner's data individually. No, what you really want is a single table in which you can find the tab corresponding to a particular diner. You can then look up the tab of the diner with dining club card number 7712 in row number 7712 of the table much more easily.

This is why arrays were invented. Arrays are a convenient way to group many variables under a single variable name. They are like pigeonholes, with each compartment storing a single value. Arrays can be one-dimensional like a list, two-dimensional like a chessboard, or three-dimensional like an apartment building — in fact, they can have any arbitrary dimensionality, including ones humans cannot visualise easily.

An array is defined using square brackets `[...]`. For example: an array of three integers called `my_list` would be declared thus:

```
int my_list[3];
```

This statement would cause space for three adjacent integers to be created in memory, as in the diagram below. Notice that there is no space between the name of the array above (`my_array`) and the opening square bracket '`[`'.

```
          ------------------------------------
my_list: |           |           |            |
          ------------------------------------
```

The number in the square brackets of the declaration is referred to as the *subscript* of the array, and it must be an integer greater than or equal to zero.

The three integer "pigeonholes" in the above array are called its *locations*, and the values filling them are called the array's *elements*. The position of an element in the array is called its *index* (the plural is *indices*). In the following example, 5, 17, and 23 are the array's elements, and 0, 1, and 2 are its corresponding indices.

Notice also that although we are creating space for three integers, arrays in C are *zero-based*, so the indices of the array run (0, 1, 2). If arrays in C were *one-based*, the indices would run (1, 2, 3).

```
int my_list[3];
my_list[0] = 5;
my_list[1] = 17;
my_list[2] = 23;
```

The above example would result in an array that "looks like" the following diagram. (Of course, an array is merely an arrangement of bytes in the computer's memory, so it does not *look like* much of anything, literally speaking.)

```
index:        0           1           2
          ------------------------------------
my_list: |     5      |     17     |     23      |
          ------------------------------------
```

Note that every element in an array must be of the same type, for example, integer. It is not possible in C to have arrays that contain multiple data types. However, if you want an array with multiple data types, you might instead be able to use multiple arrays of different data types that contain the same number of elements. For example, to continue our restaurant tab example above, one array, `diner_names` might contain a list of the names of the diners. If you are looking for a particular diner, say Xavier Nougat, you might find that the index of his name in `diner_names` is 7498. If you have programmed an associated floating-point array called `diner_tabs`, you might look up element 7498 in that array and find that his tab is $99.34.

14.1 Array bounds

In keeping with C's free-wheeling, "I assume you know what you're doing" policy, the compiler does not complain if you try to write to elements of an array that do not exist. For example, the code below defines an array with five elements. (Remember, C arrays are zero-based.)

```
char my_array[4];
```

Given the line of code below, your program will happily try to write the character '`*`' at location 10000. Unfortunately, as may happen when writing to an uninitialized pointer, this may crash the program, but will probably do nothing worse on a GNU system. (See Section 9.3 [Pointers and initialization], page 49.)

```
my_array[10000] = '*';
```

The first and last positions in an array are called its *bounds*. Remember that the bounds of an array are zero and the integer that equals the number of elements it contains, minus one.

Although C will not warn you at compile-time when you exceed the bounds of an array, the debugger can tell you at run-time. See Section 23.5 [Introduction to GDB], page 230, for more information.

14.2 Arrays and `for` loops

When you declare an array, the computer allocates a block of memory for it, but the block contains garbage (random values). Therefore, before using an array, you should initialise it. It is usually a good idea to set all elements in the array to zero.

The easiest way to initialise an array is with a `for` loop. The following example loops through every element in the array `my_array` and sets each to zero.

Remember, because arrays in C are zero-based, the indices of the array `my_array` in the example below run 0 through 9, rather than 1 through 10. The effect is the same, however: an array of `ARRAY_SIZE` (that is, 10) elements.

```
#include <stdio.h>
#define ARRAY_SIZE 10

int main ()
{
  int index, my_array[ARRAY_SIZE];

  for (index = 0; index < ARRAY_SIZE; index++)
    {
      my_array[index] = 0;
      printf ("my_array[%d] = %d\n", index, my_array[index]);
    }
  printf("\n");

  return 0;
}
```

The output from the above example is as follows:

```
my_array[0] = 0
my_array[1] = 0
my_array[2] = 0
my_array[3] = 0
my_array[4] = 0
my_array[5] = 0
my_array[6] = 0
my_array[7] = 0
my_array[8] = 0
my_array[9] = 0
```

You can use similar code to fill the array with different values. The following code example is nearly identical to the one above, but the line `my_array[index] = index;` fills each element of the array with its own index:

```
#include <stdio.h>
#define ARRAY_SIZE 5

int main ()
{
  int index, my_array[ARRAY_SIZE];

  for (index = 0; index < ARRAY_SIZE; index++)
    {
      my_array[index] = index;
      printf ("my_array[%d] = %d\n", index, my_array[index]);
    }
  printf("\n");

  return 0;
}
```

The output is as follows:

```
my_array[0] = 0
my_array[1] = 1
my_array[2] = 2
my_array[3] = 3
my_array[4] = 4
```

Here is a human's-eye view of the *internal representation* of the array (how the array "looks" to the computer):

```
index          0   1   2   3   4
              -------------------
element      | 0 | 1 | 2 | 3 | 4 |
              -------------------
```

You can use loops to do more than initialize an array. The next code example demonstrates the use of `for` loops with an array to find prime numbers. The example uses a mathematical device called the Sieve of Erastosthenes. Erastosthenes of Cyrene discovered that one can find all prime numbers by first writing down a list of integers from 2 (the first prime number) up to some arbitrary number, then deleting all multiples of 2 (which are by definition not prime numbers), finding the next undeleted number after 2 (which is 3), deleting all its multiples, finding the next undeleted number after that (5), deleting all its multiples, and so on. When you have finished this process, all numbers that remain are primes.

The following code example creates a Sieve of Erastosthenes for integers up to 4999, initializes all elements with 1, then deletes all composite (non-prime) numbers by replacing the elements that have an index equal to the composite with the macro `DELETED`, which equals 0.

```
#include <stdio.h>

#define ARRAY_SIZE 5000
#define DELETED    0

int sieve[ARRAY_SIZE];


int main ()
{
  printf ("Results of Sieve of Erastosthenes:\n\n");

  fill_sieve();
  delete_nonprimes();
  print_primes();
}


fill_sieve ()
{
  int index;

  for (index = 2; index < ARRAY_SIZE; index++)
```

```
    sieve[index] = 1;
}


delete_nonprimes ()
{
  int index;

  for (index = 2; index < ARRAY_SIZE; index++)
    {
      if (sieve[index] != DELETED)
delete_multiples_of_prime (index);
    }
}


delete_multiples_of_prime (int prime)
{
  int index, multiplier = 2;

  for (index = prime * multiplier; index < ARRAY_SIZE; index = prime * multiplier++)
    sieve[index] = DELETED;
}


print_primes ()
{
  int index;

  for (index = 2; index < ARRAY_SIZE; index++)
    {
      if (sieve[index] != DELETED)
printf ("%d ", index);
    }
  printf("\n\n");
}
```

Part of the output from the above program is shown below, for values up to 500. (The full output is considerably longer.)

```
Results of Sieve of Erastosthenes:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 ...
```

14.3 Multidimensional arrays

Suppose that you are writing a chess-playing program like GNU Chess (`http://www.gnu.org/software/chess/chess.html`). A chessboard is an 8-by-8 grid. What data structure would you use to represent it?

You could use an array that has a chessboard-like structure, that is, a *two-dimensional array*, to store the positions of the chess pieces. Two-dimensional arrays use two indices to pinpoint an individual element of the array. This is very similar to what is called "algebraic notation", already commonly used in chess circles to record games and chess problems.

In principle, there is no limit to the number of subscripts (or dimensions) an array can have. Arrays with more than one dimension are called *multidimensional arrays*. Although humans cannot easily visualize objects with more than three dimensions, representing multidimensional arrays presents no problem to computers.

In practice, however, the amount of memory in a computer tends to place limits on how large an array can be. For example, a simple four-dimensional array of double-precision numbers, merely twenty elements wide in each dimension, already takes up $20\hat{}4 * 8$, or 1,280,000 bytes of memory — about a megabyte. (Each element of the array takes up 8 bytes, because doubles are 64 bits wide. See Section 5.1.2 [Floating point variables], page 20, for more information.)

You can declare an array of two dimensions as follows:

variable_type array_name`[`*size1*`][`*size2*`]`

In the above example, *variable_type* is the name of some type of variable, such as `int`. Also, *size1* and *size2* are the sizes of the array's first and second dimensions, respectively. Here is an example of defining an 8-by-8 array of integers, similar to a chessboard. Remember, because C arrays are zero-based, the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same, however: a two-dimensional array of 64 elements.

```
int chessboard[8][8];
```

To pinpoint an element in this grid, simply supply the indices in both dimensions.

Every element in this grid needs two indices to pin-point it. Normally, C programmers think of element 0,0 of a two-dimensional array as being the upper-left corner. The computer, however, knows nothing of left and right, and for our purposes (attempting to conform to international chess notation), it makes more sense to mentally "flop" the array vertically so that element 0,0 is the lower-left corner of the board, and 7,7 the upper-right. Thus, the first index gives the row number for the grid and the second index gives the column number. For example, 1,0 is the square directly above the lower-left corner. Suppose that a value of 1 for an array location means a the chess king is on the space in question. To indicate the White king's usual position (that is, square a5 in algebraic chess notation or 0,4 in our zero-based integer notation), you would write this:

```
chessboard[0][4] = 1;
```

Since computer memory is essentially one-dimensional, with memory locations running straight from 0 up through the highest location in memory, a multidimensional array cannot be stored in memory as a grid. Instead, the array is dissected and stored in rows. Consider the following two-dimensional array.

```
          -----------
row 0   | 1 | 2 | 3 |
          -----------
row 1   | 4 | 5 | 6 |
          -----------
row 2   | 7 | 8 | 9 |
          -----------
```

Note that the numbers inside the boxes are not the actual indices of the array, which is two-dimensional and has two indices for each element, but only arbitrary placeholders to enable you to see which elements correspond in the following example. The row numbers do correspond to the first index of the array, so they are numbered from 0 to 2 rather than 1 to 3.

To the computer, the array above actually "looks" like this:

```
------------------------------------
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
------------------------------------
|   row 0   |   row 1   |   row 2   |
```

Another way of saying that arrays are stored by row is that the second index *varies fastest*. A two-dimensional array is always thought of as follows:

array_name`[`*row*`][`*column*`]`

Every row stored will contain elements of many columns. The column index runs from 0 to *size* - 1 inside every row in the one-dimensional representation (where *size* is the number of columns in the array), so the column index is changing faster than the row index, as the one-dimensional representation of the array inside the computer is traversed.

You can represent a three-dimensional array, such as a cube, in a similar way:

variable_type *array_name*`[`*size1*`][`*size2*`][`*size3*`]`

Arrays do not have to be shaped like squares and cubes; you can give each dimension of the array a different size, as follows:

```
int non_cube[2][6][8];
```

Three-dimensional arrays (and higher) are stored in the same basic way as two-dimensional ones. They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest (then the next-to-last, and so on).

14.4 Arrays and nested loops

To initialize multidimensional arrays, you can use nested `for` loops. Three nested loops are needed to initialize a three-dimensional array:

```
#include <stdio.h>

#define SIZE1 3
#define SIZE2 3
#define SIZE3 3

int main ()
{
  int fast, faster, fastest;
  int my_array[SIZE1][SIZE2][SIZE3];

  for (fast = 0; fast < SIZE1; fast++)
    {
      for (faster = 0; faster < SIZE2; faster++)
{
  for (fastest = 0; fastest < SIZE3; fastest++)
    {
      my_array[fast][faster][fastest] = 0;
      printf("my_array[%d][%d][%d] DONE\n", fast, faster, fastest);
    }
}
    }
  printf("\n");
}
```

In this example, the variables `fast`, `faster`, and `fastest` contain the indices of the array, and vary fast, faster, and fastest, respectively. In the example output below, you can see that the `fastest` index changes every line, while the `faster` index changes every three lines, and the `fast` index changes only every nine lines.

```
my_array[0][0][0] DONE
my_array[0][0][1] DONE
my_array[0][0][2] DONE
my_array[0][1][0] DONE
my_array[0][1][1] DONE
my_array[0][1][2] DONE
my_array[0][2][0] DONE
my_array[0][2][1] DONE
my_array[0][2][2] DONE
my_array[1][0][0] DONE
my_array[1][0][1] DONE
my_array[1][0][2] DONE
my_array[1][1][0] DONE
my_array[1][1][1] DONE
my_array[1][1][2] DONE
my_array[1][2][0] DONE
my_array[1][2][1] DONE
my_array[1][2][2] DONE
my_array[2][0][0] DONE
my_array[2][0][1] DONE
my_array[2][0][2] DONE
```

```
my_array[2][1][0] DONE
my_array[2][1][1] DONE
my_array[2][1][2] DONE
my_array[2][2][0] DONE
my_array[2][2][1] DONE
my_array[2][2][2] DONE
```

Note: Although in this example we have followed the order in which indices vary inside the computer, you do not have to do so in your own code. For example, we could have switched the nesting of the innermost `fastest` and outermost `fast` loops, and every element would still have been initialized. It is better, however, to be systematic about initializing multidimensional arrays.

14.5 Initializing arrays

As mentioned above, you must initialize your arrays or they will contain garbage. There are two main ways to do so. The first is by assigning values to array elements individually, either as shown in the example below, or with `for` loops. (See Section 14.2 [Arrays and for loops], page 90, above.)

```
my_array[0] = 42;
my_array[1] = 52;
my_array[2] = 23;
my_array[3] = 100;
...
```

The second method is more efficient and less tedious. It uses a single assignment operator (`=`) and a few curly brackets (`{...}`).

Recall that arrays are stored by row, with the last index varying fastest. A 3 by 3 array could be initialized in the following way:

```
int my_array[3][3] =
{
  {10, 23, 42},
  {1, 654, 0},
  {40652, 22, 0}
};
```

Here is a small program that uses the above initialization:

```
#include <stdio.h>

int main()
{
  int row, column;

  int my_array[3][3] =
  {
    {10, 23, 42},
    {1, 654, 0},
    {40652, 22, 0}
  };

  for (row = 0; row <=2; row++)
    {
      for (column = 0; column <= 2; column++)
{
  printf("%d\t", my_array[row][column]);
}
      printf("\n");
    }

  printf("\n");

  return 0;
}
```

The internal curly brackets are unnecessary, but they help to distinguish the rows of the array. The following code has the same effect as the first example:

```
int my_array[3][3] =
{
  10, 23, 42,
  1, 654, 0,
  40652, 22, 0
};
```

The same array initialization could even be written this way:

```
int my_array[3][3] =
  {10, 23, 42, 1, 654, 0, 40652, 22, 0};
```

Using any of these three array initializations, the program above will print the following text:

```
10      23      42
1       654     0
40652   22      0
```

Note 1: Be careful to place commas after every array element except the last one before a closing curly bracket ('}'). Be sure you also place a semicolon after the final curly bracket of an array initializer, since here curly brackets are not delimiting a code block.

Note 2: All the expressions in an array initializer must be constants, not variables; that is, values such as `235` and `'q'` are acceptable, depending on the type of the array, but expressions such as the integer variable `my_int` are not.

Note 3: If there are not enough expressions in the array initializer to fill the array, the remaining elements will be set to 0 if the array is static, but will be filled with garbage otherwise.

14.6 Arrays as Parameters

There will be times when you want to pass an array as a parameter to a function. (For example, you might want to pass an array of numbers to a function that will sort them.)

In the following example, notice how the array `my_array` in `main` is passed to the function `multiply` as an actual parameter with the name `my_array`, but that the formal parameter in the `multiply` function is defined as `int *the_array`: that is, an integer pointer. This is the basis for much that you will hear spoken about the "equivalence of pointers and arrays" — much that is best ignored until you have more C programming experience. The important thing to understand is that arrays passed as parameters are considered to be pointers by the functions receiving them. Therefore, they are always variable parameters, which means that other functions can modify the original copy of the variable, just as the function `multiply` does with the array `my_array` below. (See Chapter 8 [Parameters], page 39.)

```
#include <stdio.h>

void multiply (int *, int);


int main()
{
  int index;
  int my_array[5] = {0, 1, 2, 3, 4};

  multiply (my_array, 2);

  for (index = 0; index < 5; index++)
    printf("%d  ", my_array[index]);

  printf("\n\n");
  return 0;
}


void multiply (int *the_array, int multiplier)
{
  int index;
  for (index = 0; index < 5; index++)
    the_array[index] *= multiplier;
}
```

Even though the function `multiply` is declared `void` and therefore does not return a result, it can still modify `my_array` directly, because it is a variable parameter. Therefore, the result of the program above is as follows:

```
0  2  4  6  8
```

If you find the interchangeability of arrays and pointers as formal parameters in function declarations to be confusing, you can always avoid the use of pointers, and declare formal parameters to be arrays, as in the new version of the `multiply` function below. The result is the same.

```
void multiply (int the_array[], int multiplier)
{
  int index;
  for (index = 0; index < 5; index++)
    the_array[index] *= multiplier;
}
```

14.7 Questions for Chapter 14

1. Declare an array of type `double`, measuring 4 by 5 elements.
2. How do you pass an array as a parameter?
3. When an array parameter is received by a function, does C allocate space for a local variable and copy the whole array to the new location?
4. What does it mean to say that one dimension of an array "varies fastest"?
5. Which dimension of an array varies fastest, the first or the last?

15 Strings

Communication using arrays.

So far we have examined variables that can contain integers, floating-point numbers, and values that represent individual text characters. But what if you need a variable that can contain a sequence of text characters, such as a name in a database of diners at a restaurant, as in the examples for last chapter? That's where strings and string variables come in.

A *string value* is a sequence of text characters that can become a value for a *string variable*. Both a string value and a string variable can be referred to as a *string*, depending on context.

In C, a string value is represented by text characters enclosed by double quotes:

```
"This is a string value."
```

A string can contain any character, including special control characters, such as the tab character '`\t`', the newline character '`\n`', the "bell" character '`\7`' (which causes the terminal to beep when it is displayed), and so on.

We have been using string values since we introduced the `printf` command early in the book. (See Chapter 3 [The form of a C program], page 9.) To cause your terminal to beep twice, include the following statement in a C program:

```
printf("This is a string value. Beep! Beep! \7\7");
```

15.1 Conventions and declarations

Do not confuse strings in C with individual characters. By convention, individual characters are enclosed in single quotes, like this: `'a'`, and have the variable type `char`. On the other hand, string values are enclosed in double quotes, like this: `"abcdefg"`. String variables are either arrays of type `char` or have the type "pointer to `char`", that is, `char *`.

Conceptually, a string is an array of characters (type `char`). In C, string variables can theoretically be of any length, unlike languages such as Pascal where strings hold a maximum of 255 characters. However, the length of the string value is determined by the position of the first null character (`'/0'`) in the string. Even though a string *variable* might be 32,000 characters long, or longer, if the null character first appears at position 5 in the character array, the string value is considered to be of length 5 (and contains the characters in positions 0 through 4, in sequence). You will rarely need to consider this end marker, as most functions in C's string library add it or remove it automatically.

15.2 Initializing strings

Initializing string variables (or character arrays) with string values is in many ways even easier than initializing other kinds of arrays. There are three main ways of assigning string constants to string variables. (A *string constant* is a string value that was typed into the source code, as opposed to one that is generated by the program or entered by the user.)

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

int main()
{
  /* Example 1 */
  char string1[] = "A string declared as an array.\n";

  /* Example 2 */
  char *string2 = "A string declared as a pointer.\n";

  /* Example 3 */
  char string3[30];
  strcpy(string3, "A string constant copied in.\n");

  printf (string1);
  printf (string2);
  printf (string3);

  return 0;
}
```

1. `char string1[] = "A string declared as an array.\n";`

   This is usually the best way to declare and initialize a string. The character array is declared explicitly. There is no size declaration for the array; just enough memory is allocated for the string, because the compiler knows how long the string constant is. The compiler stores the string constant in the character array and adds a null character (‘`\0`’) to the end.

2. `char *string2 = "A string declared as a pointer.\n";`

   The second of these initializations is a pointer to an array of characters. Just as in the last example, the compiler calculates the size of the array from the string constant and adds a null character. The compiler then assigns a pointer to the first character of the character array to the variable `string2`.

   Note: Most string functions will accept strings declared in either of these two ways. Consider the `printf` statements at the end of the example program above — the statements to print the variables `string1` and `string2` are identical.

3. `char string3[30];`

   Declaring a string in this way is useful when you don’t know what the string variable will contain, but have a general idea of the length of its contents (in this case, the string can be a maximum of 30 characters long). The drawback is that you will either have to use some kind of string function to assign the variable a value, as the next line of code does (`strcpy(string3, "A string constant copied in.\n");`), or you will have to assign the elements of the array the hard way, character by character. (See Section 15.4 [String library functions], page 104, for more information on the function `strcpy`.)

15.3 String arrays

Suppose you want to print out a screenful of text instead of a single line. You could use one long character array, interspersed with '\n' characters where you want the lines to break, but you might find it easier to use a *string array*. A string array is an array of strings, which, of course, are themselves arrays of characters; in effect, a string array is a two-dimensional character array.

Just as there are easy methods of initializing integer arrays, float arrays, strings, and so on, there is also an easy way of initializing string arrays. For example, here is a sample program which prints out a menu for a full application program. That's all it does, but you might imagine that when the user chooses '3' in the full program, the application invokes the function `calculate_bill` we examined earlier. (See Chapter 8 [Parameters], page 39.)

```
#include <stdio.h>

char *menu[] =
{
  "  -------------------------------------- ",
  " |            ++ MENU ++                |",
  " |           ~~~~~~~~~~~~               |",
  " |     (0) Edit Preferences             |",
  " |     (1) Print Charge Sheet           |",
  " |     (2) Print Log Sheet              |",
  " |     (3) Calculate Bill               |",
  " |     (q) Quit                         |",
  " |                                      |",
  " |                                      |",
  " |     Please enter choice below.       |",
  " |                                      |",
  "  -------------------------------------- "
};


int main()
{
  int line_num;

  for (line_num = 0; line_num < 13; line_num++)
    {
      printf ("%s\n", menu[line_num]);
    }

  return 0;
}
```

Notice that the string array `menu` is declared `char *menu[]`. This method of defining a two-dimensional string array is a combination of methods 1 and 2 for initializing strings from the last section. (See Section 15.2 [Initializing strings], page 101.) This is the most convenient method; if you try to define `menu` with `char menu[][]`, the compiler will return an "unspecified bounds error". You can get around this by declaring the second subscript of `menu` explicitly (e.g. `char menu[][80]`), but that necessitates you know the maximum length of the strings you are storing in the array, which is something you may not know and that it may be tedious to find out.

The elements of `menu` are initialized with string constants in the same way that an integer array, for example, is initialized with integers, separating each element with a comma. (See Section 14.5 [Initializing arrays], page 97.)

15.4 String library functions

The GNU C Library provides a number of very useful functions which handle strings. Here is a list of the more common ones. To use the functions beginning with '`ato`', you must include the header file '`stdlib.h`'; to use the functions beginning with '`str`', you must include the header file '`string.h`'.

- `atof` Converts an ASCII string to its floating-point equivalent; for example, converts '`-23.5`' to the value -23.5.

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

int main()
{
  double my_value;
  char my_string[] = "+1776.23";
  my_value = atof(my_string);
  printf("%f\n", my_value);

  return 0;
}
```

  The output from the above code is '`1776.230000`'.

- `atoi` Converts an ASCII string to its integer equivalent; for example, converts '`-23.5`' to the value -23.

```
int my_value;
char my_string[] = "-23.5";
my_value = atoi(my_string);
printf("%d\n", my_value);
```

- `atol` Converts an ASCII string to its long integer equivalent; for example, converts '`+2000000000`' to the value 2000000000.

```
long my_value;
char my_string[] = "+2000000000";
my_value = atol(my_string);
printf("%ld\n", my_value);
```

- `strcat` *Concatenates* two strings: that is, joins them together into one string. Example:

```
char string1[50] = "Hello, ";
char string2[] = "world!\n";
strcat (string1, string2);
printf (string1);
```

The example above attaches the contents of `string2` to the current contents of `string1`. The array `string1` then contains the string '`Hello, world!\n`'.

Notice that `string1` was declared to be 50 characters long, more than enough to contain the initial values of both `string1` and `string2`. You must be careful to allocate enough space in the string variable that will receive the concatenated data; otherwise, your program is likely to crash. Again, on a GNU system, although your program won't run, nothing more drastic than an error message from the operating system is likely to occur in such a case.

- `strcmp` Compares two strings and returns a value that indicates which string comes first alphabetically. Example:

```
int comparison;
char string1[] = "alpha";
char string2[] = "beta";

comparison = strcmp (string1, string2);
printf ("%d\n", comparison);
```

If the two strings are identical, `strcmp` returns 0. If the first string passed to `strcmp` comes alphabetically before the second (that is, the first string is "less than" the second one), `strcmp` returns a value less than 0. If the first string comes alphabetically after the second one (that is, the first string is "greater than" the second one), `strcmp` returns a value greater than zero. (Note that numbers come before letters in ASCII, and upper-case letters come before lower-case ones.)

The example above prints out '`-1`', because '`alpha`' is alphabetically "less than" '`beta`'.

In all cases below, `string1` comes alphabetically before `string2`, so `strcmp(string1, string2)` returns a negative value.

'`string1`'	`string2`
'`aaa`'	'`aab`'
'`aaa`'	'`aaba`'
'`aa`'	'`aaa`'

- `strcpy` Copies a string into a string variable. Example:

```
char dest_string[50];
char source_string[] = "Are we not men?";

/* Example 1 */
strcpy (dest_string, source_string);
printf ("%s\n", dest_string);

/* Example 2 */
strcpy (dest_string, "Are we having fun yet?");
printf ("%s\n", dest_string);
```

The example above produces this output:

```
Are we not men?
Are we having fun yet?
```

Notes:

The destination string in strcmp comes first, then the source string. This works in exactly the opposite way from the GNU/Linux shell command, cp.

You can use strcmp to copy one string variable into another (Example 1), or to copy a string constant into a string variable (Example 2).

Note the use of the characters '%s' in the printf statements to display a string, rather than '%d' to display an integer or '%f' to display a float.

- strlen Returns an integer that gives the length of a string in characters, not including the null character at the end of the string. The following example displays the number '5'.

```
int string_length
char my_string[] = "fnord";

string_length = strlen (my_string);
printf ("%d\n", string_length);
```

- strncat Works like strcat, but concatenates only a specified number of characters. The example below displays the string 'Hello, world! Bye'.

```
char string1[50] = "Hello, world! ";
char string2[] = "Bye now!";
strncat (string1, string2, 3);
printf ("%s\n", string1);
```

- strncmp Works like strcmp, but compares only a specified number of characters of both strings. The example below displays '0', because 'dogberry' and 'dogwood' are identical for their first three characters.

```
int comparison;
char string1[] = "dogberry";
char string2[] = "dogwood";

comparison = strncmp (string1, string2, 3);
printf ("%d\n", comparison);
```

- strncpy Works like strcpy, but copies only a specified number of characters. The example below displays the string 'Are we', because only the first six characters of source_string are being copied into dest_string.

```
char dest_string[50];
char source_string[] = "Are we not men?";

strncpy (dest_string, source_string, 6);
printf ("%s\n", dest_string);
```

Note: As in strcmp, the destination string in strncmp comes first, then the source string. This works in exactly the opposite way from the GNU/Linux shell command, cp.

- strstr Tests whether a substring is present in a larger string. Returns a pointer to the first occurrence of the substring in the larger string, or zero if the substring is not present. (When the substring is empty, strstr returns a pointer to the first character of the larger string.)

  The example below displays ''foo' is a substring of 'Got food?'.'.

```
char string1[] = "Got food?";
char string2[] = "foo";

if (strstr (string1, string2))
  printf("'%s' is a substring of '%s'.\n", string2, string1);
```

15.5 Questions for Chapter 15

1. What are the three main ways of initializing string variables?
2. How would you declare an array of strings?
3. What information is returned by `strlen`?
4. What does the function `strcat` do? How about `strncat`?
5. Rewrite the Morse coder program more efficiently, using static strings. (See Section 10.6 [Example 15], page 58, for the original version.)

16 Input and output

Input and output. Talking to the user. Why your printer is a file.

In order for a program to do anything useful, it usually must do some kind of input and output, whether input from the keyboard and output to the screen, or input from and output to the computer's hard disk. While the C language itself does not provide much in the way of input and output functions, the GNU C Library contains so many facilities for input and output that a whole book could be written about them. In this chapter, we will focus on the basics. For more information on the functions described in this chapter, and many more, we urge you to consult ⟨undefined⟩ [Table of Contents], page ⟨undefined⟩.

Most objects from which you can receive input and to which you can send output on a GNU system are considered to be files — not only are files on your hard disk (such as object code files, C source code files, and ordinary ASCII text files) considered to be files, but also such peripherals as your printer, your keyboard, and your computer monitor. When you write a C program that prompts the user for input from the keyboard, your program is *reading from*, or accepting input from, the keyboard, in much the same way that it would read a text string from a text file. Similarly, when your C program displays a text string on the user's monitor, it is *writing to*, or sending output to, the terminal, just as though it were writing a text string to a text file. In fact, in many cases you'll be using the very same functions to read text from the keyboard and from text files, and to write text to the terminal and to text files.

This curious fact will be explored later in the chapter. For now it is sufficient to say that when C treats your computer's peripherals as files, they are known as *devices*, and each one has its own name, called a *device name* or *pseudo-device name*. On a GNU system, the printer might be called '`/dev/lp0`' (for "device line printer zero") and the first floppy drive might be called '`/dev/fd0`' (for "device floppy drive zero"). (Why zero in both cases? Most objects in the GNU environment are counted by starting with zero, rather than one — just as arrays in C are zero-based.)

The advantage of treating devices as files is that it is often not necessary to know how a particular device works, only that it is connected to the computer and can be written to or read from. For example, C programs often get their input from the keyboard, which C refers to with the file name '`stdin`' (for "standard input"), and C programs often send their output to the monitor's text display, referred to as '`stdout`'. In some cases, '`stdin`' and '`stdout`' may refer to things other than the keyboard and monitor; for example, the user may be redirecting the output from your program to a text file with the `>` command in GNU/Linux. The beauty of the way the standard input/output library handles things is that your program will work just the same.

Before you can read from or write to a file, you must first connect to it, or *open* it, usually by either the `fopen` command, which returns its stream, or the `open` command, which returns its file descriptor. You can open a file for reading, writing, or both. You can also open a file for *appending*, that is, writing data after the current end of the file.

Files are made known to functions not by their file names, except in a few cases, but by identifiers called "streams" or "file descriptors". For example, `printf` uses a stream as an identifier, not the name of the file. So does `fclose`:

```
fprintf (my_stream, "Just a little hello from fprintf.\n");
close_error = fclose (my_stream);
```

On the other hand, `fopen` takes a name, and returns a stream:

```
my_stream = fopen (my_filename, "w");
```

This is how you map from names to streams or file descriptors: you open the file (for reading, writing, or both, or for appending), and the value returned from the `open` or `fopen` function is the appropriate file descriptor or stream.

You can operate on a file either at a high level or at a low level. Operating on a file at a high level means that you are using the file at a high level of abstraction. (See Chapter 1 [Introduction], page 1, to refresh your memory about the distinction between high and low levels of abstraction.) Using high-level functions is usually safer and more convenient than using low-level functions, so we will mostly concern ourselves with high-level functions in this chapter, although we will touch on some low-level functions toward the end.

A high-level connection opened to a file is called a *stream*. A low-level connection to a file is called a *file descriptor*. Streams and file descriptors have different data types, as we shall see. You must pass either a stream or a file descriptor to most input/output functions, to tell them which file they are operating on. Certain functions (usually high-level ones) expect to be passed streams, while others (usually low-level ones) expect file descriptors. A few functions will accept a simple filename instead of a stream or file descriptor, but generally these are only the functions that initialize streams or file descriptors in the first place.

You may consider it a nuisance to have to use a stream or a file descriptor to access your file when a simple file name would seem to suffice, but these two mechanisms allow a level of abstraction to exist between your code and your files. Remember the "black box" analogy we explored at the beginning of the book. By using the data in files only through streams or file descriptors, we are guaranteed the ability to write a rich variety of functions that can exploit the behavior of these two "black box" abstractions.

Interestingly enough, although streams are considered to be for "high-level" input/output, and file descriptors for "low-level" I/O, and GNU systems support both, more Unix-like systems support streams than file descriptors. You can expect any system running ISO C to support streams, but non-GNU systems may not support file descriptors at all, or may only implement a subset of the GNU functions that operate on file descriptors. Most of the file descriptor functions in the GNU library are included in the POSIX.1 standard, however.

Once you have finished your input and output operations on the file, you must terminate your connection to it. This is called *closing* the file. Once you have closed a file, you cannot read from or write to it anymore until you open it again.

In summary, to use a file, a program must go through the following routine:

- Open the file for reading, writing, or both.
- Read from or write to the file as appropriate, using file-handling functions provided by the GNU C Library.
- Close the file

16.1 High-level file routines

You can recognise most of the high-level input/output functions that operate on files because they begin with the letter '`f`'; for example, the high-level function for opening a file called `fopen`, as opposed to the low-level file-opening function `open`. Some of them are more generalized versions of functions with which you may already be familiar; for example, the function `fprintf` behaves like the familiar `printf`, but takes an additional parameter — a stream — and sends all its output to that stream instead of simply sending its output to '`stdout`', as `printf` does.

16.1.1 Opening a file

The main high-level function for opening files is `fopen`. When you open a file with the `fopen` function, the GNU C Library creates a new stream and creates a connection between the stream and a file. If you pass this function the name of a file that does not exist, that file will be created. The `fopen` function normally returns a stream. A stream is a flow of data from a source to a destination within a GNU system. Programs can read characters from or write characters to a stream without knowing either the source or destination of the data, which may be a file on disk, a device such as a terminal meant as a human interface, or something entirely different. Streams are represented by variables of type `FILE *` — `fopen` will return a null pointer if it fails to open the file.

The first parameter to this function is a string containing the filename of the file to open. The filename string can be either a constant or a variable, as in the following two equivalent examples:

```
FILE *my_stream;
my_stream = fopen ("foo", "r");

FILE *my_stream;
char my_filename = "foo";
my_stream2 = fopen (my_filename, "r");
```

The second parameter is a string containing one of the following sets of characters:

`r` Open the file for reading only. The file must already exist.

`w` Open the file for writing only. If the file already exists, its current contents are deleted. If the file does not already exist, it is created.

`r+` Open the file for reading and writing. The file must already exist. The contents of the file are initially unchanged, but the file position is set to the beginning of the file.

`w+` Open the file for both writing and reading. If the file already exists, its current contents are deleted. If the file does not already exist, it is created.

`a` Open the file for appending only. Appending to a file is the same as writing to it, except that data is only written to the current end of the file. If the file does not already exist, it is created.

`a+` Open the file for both appending and reading. If the file exists, its contents are unchanged until appended to. If the file does not exist, it is created. The initial file position for reading is at the beginning of the file, but the file position for appending is at the end of the file.

See Section 16.1.4 [File position], page 116, for more information on the concept of file position.

You can also append the character '`x`' after any of the strings in the table above. This character causes `fopen` to fail rather than opening the file if the file already exists. If you append '`x`' to any of the arguments above, you are guaranteed not to *clobber* (that is, accidentally destroy) any file you attempt to open. (Any other characters in this parameter are ignored on a GNU system, but may be meaningful on other systems.)

The following example illustrates the proper use of `fopen` to open a text file for reading (as well as highlighting the fact that you should clean up after yourself by closing files after you are done with them). Try running it once, then running it a second time after creating the text file '`snazzyjazz.txt`' in the current directory with a GNU command such as `touch snazzyjazz.txt`.

```
#include <stdio.h>

int main()
{
  FILE *my_stream;

  my_stream = fopen ("snazzyjazz.txt", "r");
  if (my_stream == NULL)
    {
      printf ("File could not be opened\n");
    }
  else
    {
      printf ("File opened!  Closing it now...\n");
      /* Close stream; skip error-checking for brevity of example */
      fclose (my_stream);
    }
  return 0;
}
```

See Section 16.1.2 [Closing a file], page 112, for more information on the function `fclose`.

16.1.2 Closing a file

The basic high-level function for closing files is `fclose`. Simply pass this function a stream, and `fopen` will close it and break the connection to the corresponding file, first reading any buffered input and writing any buffered output. If the file was closed successfully, `fclose` will return 0; otherwise, it will return `EOF`.

It is important to check for errors when you close a stream to which you are writing. For example, when `fclose` attempts to write the output remaining in its buffer, it might generate an error because the disk is full. Following is an example of closing a file with write access, while checking for errors.

```
#include <stdio.h>

int main()
{
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  int close_error;

  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "Just a little hello from fprintf.\n");

  close_error = fclose (my_stream);

  if (close_error != 0)
    {
      printf ("File could not be closed.\n");
    }
  else
    {
      printf ("File closed.\n");
    }

  return 0;
}
```

16.1.3 Block input and output

You can use the two functions in this section, `fread` and `fwrite`, to read and write text in blocks of fixed size, rather than by line or character. You can also use these two functions to read and write blocks of binary data. This feature is useful if you want to read and write data in the same format used by your program. For example, you can store an entire multidimensional array of floating-point variables in a file with the `fwrite` command, then read it back in directly later with the `fread` command, without any loss of precision caused by converting the floats to strings for use with the `fprintf` function, for example. (The main drawback to using binary files rather than formatted ASCII text files is that you cannot easily read and edit the files you create with a text editor.)

For example, to write an array called `my_array`, containing `object_count` data objects (such as integers), each of size `object_size`, to the stream `my_stream`, you might use the following line of code:

```
fwrite (&my_array, object_size, object_count, my_stream);
```

To read `my_array` back from `my_stream`, you might then use the following line:

```
fread (&my_array, object_size, object_count, my_stream);
```

Here is a short table to help you remember the directions in which `fwrite` and `fread` work.

`fwrite` from an array, to a file

`fread` from a file, to an array

The fwrite function takes four parameters. The first parameter to fwrite is a void pointer (void *) to an array that contains the data that will be written to the file. The second parameter is a variable of type size_t specifying the size of each object to be written, and the third parameter, also of type size_t, specifies the number of those objects that are to be written. The final parameter is the stream to be written to (type FILE *). If the value returned by fopen does not match the third parameter passed (that is, the number of objects to be written), then there was an error.

Like fwrite, the fread function takes four parameters. Its first parameter is a void pointer to the array that will be written to. Its second parameter, of type size_t, specifies how large each object to be read is, and its third parameter, of type size_t, specifies how many of each object is to be read. The last parameter is simply the stream to read from. Again, if the return value of this function does not match the third parameter, which specifies how many object were to be read, there was an error.

Here is an example that creates an array and fills it with multiples of 2, prints it out, writes the array's data to a file with fwrite, zeroes the array and prints it out, reads the data from the file back into the array with fread, then prints the array out again so you can compare its data with the first set of data.

```
#include <stdio.h>

int main()
{
  int row, column;
  FILE *my_stream;
  int close_error;
  char my_filename[] = "my_numbers.dat";
  size_t object_size = sizeof(int);
  size_t object_count = 25;
  size_t op_return;

  int my_array[5][5] =
  {
    2,  4,  6,  8, 10,
    12, 14, 16, 18, 20,
    22, 24, 26, 28, 30,
    32, 34, 36, 38, 40,
    42, 44, 46, 48, 50
  };
  printf ("Initial values of array:\n");
  for (row = 0; row <= 4; row++)
    {
      for (column = 0; column <=4; column++)
{
  printf ("%d  ", my_array[row][column]);
}
      printf ("\n");
    }
```

```
  my_stream = fopen (my_filename, "w");
  op_return = fwrite (&my_array, object_size, object_count, my_stream);
  if (op_return != object_count)
    {
      printf ("Error writing data to file.\n");
    }
  else
    {
      printf ("Successfully wrote data to file.\n");
    }

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  printf ("Zeroing array...\n");
  for (row = 0; row <= 4; row++)
    {
      for (column = 0; column <=4; column++)
{
  my_array[row][column] = 0;
  printf ("%d  ", my_array[row][column]);
}
      printf ("\n");
    }

  printf ("Now reading data back in...\n");
  my_stream = fopen (my_filename, "r");
  op_return = fread (&my_array, object_size, object_count, my_stream);
  if (op_return != object_count)
    {
      printf ("Error reading data from file.\n");
    }
  else
    {
      printf ("Successfully read data from file.\n");
    }
  for (row = 0; row <= 4; row++)
    {
      for (column = 0; column <=4; column++)
{
  printf ("%d  ", my_array[row][column]);
}
      printf ("\n");
    }

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  return 0;
}
```

If all goes well, the code example above will produce the following output:

```
Initial values of array:
2  4  6  8  10
12  14  16  18  20
22  24  26  28  30
32  34  36  38  40
42  44  46  48  50
Successfully wrote data to file.
Zeroing array...
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
Now reading data back in...
Successfully read data from file.
2  4  6  8  10
12  14  16  18  20
22  24  26  28  30
32  34  36  38  40
42  44  46  48  50
```

If you attempt to view the file '`my_numbers.dat`' produced by the program above with a GNU command such as `more numbers.dat`, you will see only garbage, because the information is stored in binary format, not readable by humans. After attempting to view this binary file, your terminal may continue to show only garbage and you may have to reset it. You may be able to do this with a menu option (if you are running `gnome-terminal`, for example), or you may have to type `reset` blindly.

16.1.4 File position

When a person reads a book, her "location" in the book can be specified by a page number (and even a line number or word number, at finer levels of detail). Just so, it is possible (and often useful!) to know the location, or *file position*, of a stream reading from or writing to a file. And just as we sometimes want to know which chapter a friend is currently reading in a book, or to recommend that he flip backward or forward to an interesting passage, so it is frequently useful to be able to change the current file position to access a more interesting part of the file.

At the high level of the functions in this section, GNU treats all streams as streams of characters — even binary streams like the one associated with the file '`numbers.dat`' in the example for `fread` and `fwrite`. (See Section 16.1.3 [Block input and output], page 113.) This means that the file position of any stream is a simple character count — file position 0 means that we are reading or writing the first character in the file, file position 522 means that we are reading the 523rd character, and so on. (Just as with arrays in C, file positions are zero-based.)

Not only does the file position of a stream describe where in the file the stream is reading or writing, but reading or writing on the stream advances the file position. During high-level access to a file, you can change the file position at will. Any file that permits changing the file position in an arbitrary way is called a *random-access file*. (Many years ago, the people who invented computer jargon chose the word "random" to be part of the phrase "random-access" because, from the point of view of the computer, a random-access file can be read from or written to at any location, as if at random. Of course, programmers are not working randomly; they decide where their programs should read and write. The term *RAM* for *random-access memory* comes from the same source.)

The main high-level function to tell where the current file position is, is called appropriately, `ftell`. It accepts a single parameter — a file stream — and returns a long integer representing the file position.[1] (See section "libc" in *The GNU C Library Reference Manual*, for more information.)

The main function to seek a different file position is called `fseek`. It accepts three parameters. The first parameter is the stream in question, the second is a long integer offset, and the third parameter is a constant that specifies whether the offset is relative to the beginning of the file (`SEEK_SET`), to the current file position (`SEEK_CUR`), or to the end of the file (`SEEK_END`). The `fseek` function returns 0 if the operation was successful, or a nonzero integer value otherwise. (A successful `fseek` operation also clears the end-of-file indicator (see below), and discards the results of `ungetc`. See Section 16.3.5 [ungetc], page 142.)

There is a simple macro called `rewind` that will take the file pointer back to the beginning of the file. You must simply pass it the stream that you want to rewind; it does not return a value. It is the same as calling `fseek` on the stream with an offset of 0 and a third parameter of `SEEK_SET`, except that it resets the error indicator for the stream and, as mentioned, there is no return value.

An example of these functions will not be useful until we have introduced single-character I/O. See Section 16.3.3 [getc and fgetc], page 139, if you want to read a code example that uses the `ftell`, `fseek`, and `rewind` functions.

16.1.5 Stream buffering

When you write characters to a stream, they are not usually stored in the file on a character-by-character basis as soon as they are written to the stream, but instead are accumulated in a *buffer* first, then written to the file in a block when certain conditions are met. (A buffer is an area of the computer's memory that acts as a temporary holding area for input or output.) Similarly, when you are reading characters from a stream, they are often *buffered*, or stored in a buffer first.

It's important to understand how buffering works, or you may find your programs behaving in an unexpected way, reading and writing characters when you do not expect them to. (You can bypass stream buffering entirely, however, by using low-level rather than high-level file routines. See Section 16.5 [Low-level file routines], page 145, for more information.)

There are three main kinds of buffering you should know about:

---

[1] Since the file position is a long integer, the length of a file using one of these functions cannot be any greater than the maximum value of a 32-bit long integer under GNU, plus one (since the file position is zero-based) — that is, such a file cannot be any more than 2,147,483,648 bytes, or about two gigabytes long. If you need to use longer files, you can use low-level file routines, which allow for longer files and file positions through such 64-bit functions as `lseek64`.

- **No buffering:** When you write characters to an unbuffered stream, the operating system writes them to the file as soon as possible.
- **Line buffering:** When you write characters to a line-buffered stream, the operating system writes them to the file when it encounters a newline character.
- **Full buffering:** When you write characters to a fully-buffered stream, the operating system writes them to the file in blocks of arbitrary size.

Most streams are fully buffered when you open them, and this is usually the best solution. However, streams connected to interactive devices such as terminals are line-buffered when you open them; yes, this means that '`stdin`' and '`stdout`' are line-buffered.

Having '`stdin`' and '`stdout`' be line-buffered is convenient, because most meaningful chunks of data you write to them are terminated with a newline character. In order to ensure that the data you read from or write to a fully-buffered stream shows up right away, use the `fflush` function. In the jargon, this is called *flushing* the stream. Flushing moves the characters from the buffer to the file, if they haven't already been moved. After the move, other functions can then work on the characters.[2]

To use `fflush`, simply pass the function the stream you want to flush. The `fflush` function returns 0 if successful, or the value `EOF` (which is a macro defined in the GNU C Library) if there was a write error.

Note that using `fflush` is not always necessary; output is flushed automatically when you try to write and the output buffer is already full, when the stream is closed, when the program exits, when an input operation on a stream actually reads data from the file, and of course, when a newline is written to a line-buffered stream. (See Section 16.2.1.2 [fputs], page 119, for a code example that uses `fflush`.)

16.1.6 End-of-file and error functions

If a file has been read to its end (that is, the current file position is the end of the file), a flag indicating this, called the *end-of-file indicator*, will be set to `TRUE`. You can check whether the end-of-file indicator has been set (and therefore whether the current file position is the end of the file), with the `feof` function. This function takes a single argument (a stream), and returns `TRUE` (a nonzero value) if the end of the file has been reached, and `FALSE` (zero) otherwise.

Another flag, the *error indicator*, indicates whether an error has occurred during an earlier operation on the stream. It returns `TRUE` if there has been an error, and `FALSE` otherwise. You can check the error indicator for a stream with the `ferror` function. This function takes a single argument (a stream), and returns `TRUE` (a nonzero value) if an error has occured during an operation on the stream, and `FALSE` (zero) otherwise.

Unfortunately, `ferror` will not tell you what the error was, or when it occurred, only whether there has been an error. To get a more detailed diagnosis, you can check the global system variable `errno`. (See Section 16.5.1 [Usual file name errors], page 146.)

[2] Strictly speaking, there are multiple levels of buffering on a GNU system. Even after flushing characters to a file, data from the file may remain in memory, unwritten to disk. On GNU systems, there is an independently-running system program, or *daemon*, that periodically commits relevant data still in memory to disk. Under GNU/Linux, this daemon is called '`bdflush`'.

It is possible to reset the error and end-of-file indicators once they have been set for a stream. To do so, simply pass the stream to the function `clearerr`; this will set both the error and end-of-file indicators back to 0. The `clearerr` function does not return a value.

You should not simply reset the error flag and try a stream operation that failed a second time. Because of buffering, you may lose or repeat data when writing, or access the wrong part of the file when reading. Before you try a failed stream operation again, you should seek to a known file position. (See Section 16.1.4 [File position], page 116.) However, most errors cannot be recovered from anyway — trying the operation again will likely result in the same error — so it is probably better to have your program report the error to the user and exit than to write complicated error-recovery routines for stream operation.

An example of these functions will not be useful until we have introduced single-character I/O. See Section 16.3.3 [getc and fgetc], page 139, if you want to read a code example that uses the `feof` and `ferror` functions.

16.2 String output and input

We will now examine some high-level file functions for reading strings from and writing strings to streams. The two string output methods we will examine (`puts` and `fputs`) are very safe to use, but the input methods run from the antiquated and very dangerous `gets` to the safer `fgets`, to `getline` and `getdelim`, two GNU-specific extensions to the C language that are extremely safe to use.

It is important to use the safer and better GNU functions when you can. However, you will probably still want to learn how to read and understand older (but still free) code that is unsafe (perhaps to update it and make it safe), so this book describes functions like `gets` despite the fact that they are unsafe.

16.2.1 Unformatted string output

The functions in this section are for output of strings to streams. They are generally very safe to use.

16.2.1.1 puts

The most convenient function for printing a simple message on standard outout is `puts`. It is even simpler than `printf`, since you do not need to include a newline character — `puts` does that for you.

Using `puts` couldn't be simpler. Here is an example:

```
puts ("Hello, multiverse.");
```

This code example will print the string '`Hello, multiverse.`' to standard output.

The `puts` function is safe and simple, but not very flexible. See Section 16.2.2 [Formatted string output], page 120, if you want to print fancier output.

16.2.1.2 fputs

The `fputs` ("file put string") function is similar to the `puts` function in almost every respect, except that it accepts a second parameter, a stream to which to write the string.

It does not add a newline character, however; it only writes the characters in the string. It returns `EOF` if an error occurs; otherwise it returns a non-negative integer value.

Here is a brief code example that creates a text file and uses `fputs` to write into it the phrase '`If it's not too late... make it a cheeseburger.`', followed by a newline character. This example also demonstrates the use of the `fflush` function. (See Section 16.1.5 [Stream buffering], page 117, for more information on this function.)

```
#include <stdio.h>

int main()
{
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  int flush_status;

  my_stream = fopen (my_filename, "w");
  fputs ("If it's not too late... make it a cheeseburger.\n", my_stream);

  /*
     Since the stream is fully-buffered by default, not line-buffered,
     it needs to be flushed periodically.  We'll flush it here for
     demonstration purposes, even though we're about to close it.
  */
  flush_status = fflush (my_stream);
  if (flush_status != 0)
    {
      puts ("Error flushing stream!");
    }
  else
    {
      puts ("Stream flushed.");
    }


  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  return 0;
}
```

16.2.2 Formatted string output

The functions in this section are for formatted output of strings to streams. They are generally quite safe to use.

Formatted output is textual output via functions such as `printf` or `fprintf`. These take as an argument a string containing special character sequences such as '`%d`' (which indicates that an integer argument will follow). After this string, other arguments that correspond to the special character sequences follow. When the functions combine these arguments, the result is formatted textual output.

The next several sections discuss four formatted output functions. The most basic, `printf`, prints to standard output. The `fprintf` function is a high-level routine that sends its output to a stream, `sprintf` "prints" to a string, and `asprintf` is a safer way of printing to a string.

16.2.2.1 printf

If you have been reading the book closely up to this point, you have seen the use of the `printf` function many times. To recap, this function prints a text string to the terminal (or, to be more precise, the text stream '`stdout`'). For example, the following line of code prints the string '`Hello there!`', followed by a newline character, to the console:

```
printf ("Hello there!\n");
```

You probably also remember that you can incorporate numeric constants and variables into your strings. Consider the following code example:

```
printf ("I'm free!  I'm free!  (So what?  I'm %d.)\n", 4);
```

The previous example is equivalent to the following one:

```
int age = 4;
printf ("I'm free!  I'm free!  (So what?  I'm %d.)\n", age);
```

Both of the code examples above produce the following output:

```
I'm free!  I'm free!  (So what?  I'm 4.)
```

You may recall that besides using '`%d`' with `printf` to print integers, we have also used '`%f`' on occasion to print floating-point numbers, and that on occasion we have used more than one argument. Consider this example:

```
printf ("I'm free!  I'm free!  (So what?  I'm %d.)  Well, I'm %f.\n", 4, 4.5);
```

That example produces the following output:

```
I'm free!  I'm free!  (So what?  I'm 4.)  Well, I'm 4.500000.
```

In fact, `printf` is a very flexible function. The general scheme is that you provide it with a *format string* or *template string* (such as '`"So what? I'm %d."`'), which can contain zero or more *conversion specifications*, *conversion specifiers*, or sometimes just *conversions* (in this case '`%d`'), and zero or more arguments (for example, '`4`'). Each conversion specification is said to specify a *conversion*, that is, how to convert its corresponding argument into a printable string. After the template string, you supply one argument for each conversion specifier in the template string. The `printf` function then prints the template string, including each argument as converted to a printable sub-string by its conversion specifier, and returns an integer containing the number of characters printed, or a negative value if there was an error.

16.2.2.2 Formatted output conversion specifiers

There are many different conversion specifiers that can be used for various data types. Conversion specifiers can become quite complex; for example, '`%-17.7ld`' specifies that `printf` should print the number left-justified ('`-`'), in a field at least seventeen characters wide ('`17`'), with a minimum of seven digits ('`.7`'), and that the number is a long integer ('`l`') and should be printed in decimal notation ('`%d`').

In this section, we will examine the basics of `printf` and its conversion specifiers. (For even more detail, section "Formatted Output" in *The GNU C Library Reference Manual*.)

A conversion specifier begins with a percent sign, and ends with one of the following *output conversion characters*. The most basic conversion specifiers simply use a percent sign and one of these characters, such as '`%d`' to print an integer. (Note that characters in the template string that are not part of a conversion specifier are printed as-is.)

'`c`' Print a single character.

'`d`' Print an integer as a signed decimal number.

'`e`' Print a floating-point number in exponential notation, using lower-case letters. The exponent always contains at least two digits. Example: '`6.02e23`'.

'`E`' Same as '`e`', but uses upper-case letters. Example: '`6.02E23`'.

'`f`' Print a floating-point number in normal, fixed-point notation.

'`i`' Same as '`d`'.

'`m`' Print the string corresponding to the specified value of the system `errno` variable. (See Section 16.5.1 [Usual file name errors], page 146.) GNU systems only.

'`s`' Print a string.

'`u`' Print an unsigned integer.

'`x`' Print an integer as an unsigned hexadecimal number, using lower-case letters.

'`X`' Same as '`x`', but uses upper-case letters.

'`%`' Print a percent sign ('`%`').

In between the percent sign ('`%`') and the output conversion character, you can place some combination of the following *modifiers*. (Note that the percent sign conversion ('`%%`') doesn't use arguments or modifiers.)

- Zero or more flag characters, from the following table:

  '`-`'

  Left-justify the number in the field (right justification is the default). Can also be used for string and character conversions ('`%s`' and '`%c`').

  '`+`'

  Always print a plus or minus sign to indicate whether the number is positive or negative. Valid for '`%d`', '`%e`', '`%E`', and '`%i`'.

  '`Space character`'

  If the number does not start with a plus or minus sign, prefix it with a space character instead. This flag is ignored if the '`+`' flag is specified.

  '`#`'

  For '`%e`', '`%E`', and '`%f`', forces the number to include a decimal point, even if no digits follow. For '`%x`' and '`%X`', prefixes '`0x`' or '`0X`', respectively.

'`,`'

Separate the digits of the integer part of the number into groups, using a locale-specific character. In the United States, for example, this will usually be a comma, so that one million will be rendered '`1,000,000`'. GNU systems only.

'`0`'

Pad the field with zeroes instead of spaces; any sign or indication of base (such as '`0x`') will be printed before the zeroes. This flag is ignored if the '`-`' flag or a precision is specified.

In the example given above, '`%-17.7ld`', the flag given is '`-`'.

- An optional non-negative decimal integer specifying the minimum field width within which the conversion will be printed. If the conversion contains fewer characters, it will be padded with spaces (or zeroes, if the '`0`' flag was specified). If the conversion contains more characters, it will not be truncated, and will overflow the field. The output will be right-justified within the field, unless the '`-`' flag was specified. In the example given above, '`%-17.7ld`', the field width is '`17`'.
- For numeric conversions, an optional precision that specifies the number of digits to be written. If it is specified, it consists of a dot character ('`.`'), followed by a non-negative decimal integer (which may be omitted, and defaults to zero if it is). In the example given above, '`%-17.7ld`', the precision is '`.7`'. Leading zeroes are produced if necessary. If you don't specify a precision, the number is printed with as many digits as necessary (with a default of six digits after the decimal point). If you supply an argument of zero with and explicit precision of zero, `printf` will not print any characters. Specifying a precision for a string conversion ('`%s`') indicates the maximum number of characters to write.
- An optional *type modifier character* from the table below. This character specifies the data type of the argument if it is different from the default. In the example given above, '`%-17.7ld`', the type modifier character is '`l`'; normally, the '`d`' output conversion character expects a data type of `int`, but the '`l`' specifies that a `long int` is being used instead.

  The numeric conversions usually expect an argument of either type `int`, `unsigned int`, or `double`. (The '`%c`' conversion converts its argument to `unsigned char`.) For the integer conversions ('`%d`' and '`%i`'), `char` and `short` arguments are automatically converted to type `int`, and for the unsigned integer conversions ('`%u`', '`%x`', and '`%X`'), they are converted to type `unsigned int`. For the floating-point conversions ('`%e`', '`%E`', and '`%f`'), all `float` arguments are converted to type `double`. You can use one of the type modifiers from the table below to specify another type of argument.

'`l`' Specifies that the argument is a `long int` (for '`%d`' and '`%i`'), or an `unsigned long int` (for '`%u`', '`%x`', and '`%X`').

'`L`' Specifies that the argument is a `long double` for the floating-point conversions ('`%e`', '`%E`', and '`%f`'). Same as '`ll`', for integer conversions ('`%d`' and '`%i`').

'll' Specifies that the argument is a `long long int` (for '%d' and '%i'). On systems that do not have extra-long integers, this has the same effect as 'l'.

'q' Same as 'll'; comes from calling extra-long integers "quad ints".

'z' Same as 'Z', but GNU only, and deprecated.

'Z' Specifies that the argument is of type `size_t`. (The `size_t` type is used to specify the sizes of blocks of memory, and many functions in this chapter use it.)

Make sure that your conversion specifiers use valid syntax; if they do not, if you do not supply enough arguments for all conversion specifiers, or if any arguments are of the wrong type, unpredictable results may follow. Supplying too many arguments is not a problem, however; the extra arguments are simply ignored.

Here is a code example that shows various uses of `printf`.

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

int main()
{
  int my_integer = -42;
  unsigned int my_ui = 23;
  float my_float = 3.56;
  double my_double = 424242.171717;
  char my_char = 'w';
  char my_string[] = "Pardon me, may I borrow your nose?";

  printf ("Integer: %d\n", my_integer);
  printf ("Unsigned integer: %u\n", my_ui);

  printf ("The same, as hexadecimal: %#x %#x\n", my_integer, my_ui);

  printf ("Floating-point: %f\n", my_float);
  printf ("Double, exponential notation: %17.11e\n", my_double);

  printf ("Single character: %c\n", my_char);
  printf ("String: %s\n", my_string);

  errno = EACCES;
  printf ("errno string (EACCES): %m\n");

  return 0;
}
```

The code example above produces the following output on a GNU system:

```
Integer: -42
Unsigned integer: 23
The same, as hexadecimal: 0xffffffd6 0x17
Floating-point: 3.560000
Double, exponential notation: 4.24242171717e+05
Single character: w
String: Pardon me, may I borrow your nose?
errno string (EACCES): Permission denied
```

16.2.3 fprintf

The `fprintf` ("file print formatted") command is identical to `printf`, except that its first parameter is a stream to which to send output. The following code example is the same as the one for `printf`, except that it sends its output to the text file '`snazzyjazz.txt`'.

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

int main()
{
  int my_integer = -42;
  unsigned int my_ui = 23;
  float my_float = 3.56;
  double my_double = 424242.171717;
  char my_char = 'w';
  char my_string[] = "Pardon me, may I borrow your nose?";
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "Integer: %d\n", my_integer);
  fprintf (my_stream, "Unsigned integer: %u\n", my_ui);

  fprintf (my_stream, "The same, as hexadecimal: %#x %#x\n", my_integer, my_ui);

  fprintf (my_stream, "Floating-point: %f\n", my_float);
  fprintf (my_stream, "Double, exponential notation: %17.11e\n", my_double);

  fprintf (my_stream, "Single character: %c\n", my_char);
  fprintf (my_stream, "String: %s\n", my_string);
  errno = EACCES;
  fprintf (my_stream, "errno string (EACCES): %m\n");

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  return 0;
}
```

16.2.4 asprintf

The `asprintf` (mnemonic: "allocating string print formatted") command is identical to `printf`, except that its first parameter is a string to which to send output. It terminates the string with a null character. It returns the number of characters stored in the string, not including the terminating null.

The `asprintf` function is nearly identical to the simpler `sprintf`, but is much safer, because it dynamically allocates the string to which it sends output, so that the string will never overflow. The first parameter is a pointer to a string variable, that is, it is of type `char **`. The return value is the number of characters allocated to the buffer, or a negative value if an error occurred.

The following code example prints the string '`Being 4 is cool, but being free is best of all.`' to the string variable `my_string`, then prints the string on the screen. Notice that `my_string` is not initially allocated any space at all; `asprintf` allocates the space itself. (See Section 16.2.1.1 [puts], page 119, for more information on the `puts` function.)

```
#include <stdio.h>

int main()
{
  char *my_string;

  asprintf (&my_string, "Being %d is cool, but being free is best of all.", 4);
  puts (my_string);

  return 0;
}
```

16.2.5 Deprecated formatted string output functions

This section discusses unsafe functions for formatted string output. It actually contains only one function, `sprintf`. You should never use the `sprintf` function; use `asprintf` instead.

16.2.5.1 sprintf

The `sprintf` ("string print formatted") command is similar to `asprintf`, except that it is much less safe. Its first parameter is a string to which to send output. It terminates the string with a null character. It returns the number of characters stored in the string, not including the terminating null.

This function will behave unpredictably if the string to which it is printing overlaps any of its arguments. It is dangerous because the characters output to the string may overflow it. This problem cannot be solved with the field width modifier to the conversion specifier, because only the minimum field width can be specified with it. To avoid this problem, it is better to use `asprintf`, but there is a lot of C code that still uses `sprintf`, so it is important to know about it. (See Section 16.2.4 [asprintf], page 126.)

The following code example prints the string '`Being 4 is cool, but being free is best of all.`' to the string variable `my_string` then prints the string on the screen.

Notice that `my_string` has been allocated 100 bytes of space, enough to contain the characters output to it. (See Section 16.2.1.1 [puts], page 119, for more information on the `puts` function.)

```
#include <stdio.h>

int main()
{
  char my_string[100];

  sprintf (my_string, "Being %d is cool, but being free is best of all.", 4);
  puts (my_string);

  return 0;
}
```

16.2.6 String input

The functions in this section are for input of strings from streams. They are generally very safe to use.

16.2.6.1 `getline`

The `getline` function is the preferred method for reading lines of text from a stream, including standard input. The other standard functions, including `gets`, `fgets`, and `scanf`, are too unreliable. (Doubtless, in some programs you will see code that uses these unreliable functions, and at times you will come across compilers that cannot handle the safer `getline` function. As a professional, you should avoid unreliable functions and any compiler that requires you to be unsafe.)

The `getline` function reads an entire line from a stream, up to and including the next newline character. It takes three parameters. The first is a pointer to a block allocated with `malloc` or `calloc`. (These two functions allocate computer memory for the program when it is run. See Section 20.2 [Memory allocation], page 203, for more information.) This parameter is of type `char **`; it will contain the line read by `getline` when it returns. The second parameter is a pointer to a variable of type `size_t`; this parameter specifies the size in bytes of the block of memory pointed to by the first parameter. The third parameter is simply the stream from which to read the line.

The pointer to the block of memory allocated for `getline` is merely a suggestion. The `getline` function will automatically enlarge the block of memory as needed, via the `realloc` function, so there is never a shortage of space — one reason why `getline` is so safe. Not only that, but `getline` will also tell you the new size of the block by the value returned in the second parameter.

If an error occurs, such as end of file being reached without reading any bytes, `getline` returns -1. Otherwise, the first parameter will contain a pointer to the string containing the line that was read, and `getline` returns the number of characters read (up to and including the newline, but not the final null character). The return value is of type `ssize_t`.

Although the second parameter is of type pointer to string (`char **`), you cannot treat it as an ordinary string, since it may contain null characters before the final null character marking the end of the line. The return value enables you to distinguish null characters that getline read as part of the line, by specifying the size of the line. Any characters in the block up to the number of bytes specified by the return value are part of the line; any characters after that number of bytes are not.

Here is a short code example that demonstrates how to use getline to read a line of text from the keyboard safely. Try typing more than 100 characters. Notice that getline can safely handle your line of input, no matter how long it is. Also note that the puts command used to display the line of text read will be inadequate if the line contains any null characters, since it will stop displaying text at the first null, but that since it is difficult to enter null characters from the keyboard, this is generally not a consideration.

```
#include <stdio.h>

int main()
{
  int bytes_read;
  int nbytes = 100;
  char *my_string;

  puts ("Please enter a line of text.");

  /* These 2 lines are the heart of the program. */
  my_string = (char *) malloc (nbytes + 1);
  bytes_read = getline (&my_string, &nbytes, stdin);

  if (bytes_read == -1)
    {
      puts ("ERROR!");
    }
  else
    {
      puts ("You typed:");
      puts (my_string);
    }

  return 0;
}
```

16.2.6.2 getdelim

The getdelim function is a more general form of the getline function; whereas getline stops reading input at the first newline character it encounters, the getdelim function enables you to specify other delimiter characters than newline. In fact, getline simply calls getdelim and specifies that the delimiter character is a newline.

The syntax for getdelim is nearly the same as that of getline, except that the third parameter specifies the delimiter character, and the fourth parameter is the stream from which to read. You can exactly duplicate the getline example in the last section with getdelim, by replacing the line

```
bytes_read = getline (&my_string, &nbytes, stdin);
```

with the line

```
bytes_read = getdelim (&my_string, &nbytes, '\n', stdin);
```

16.2.7 Deprecated string input functions

The functions in this section are for input of strings from streams, but they are generally dangerous and should only be called when there is no alternative. They are included here because you may come across code imported from a non-GNU system that uses these unsafe functions.

16.2.7.1 gets

If you want to read a string from standard input, you can use the `gets` function, the name of which stands for "get string". However, this function is *deprecated* — that means it is obsolete and it is strongly suggested you do not use it — because it is dangerous. It is dangerous because it provides no protection against overflowing the string into which it is saving data. Programs that use `gets` can actually be a security problem on your computer. Since it is sometimes used in older code (which is why the GNU C Library still provides it), we will examine it briefly; nevertheless, you should *always* use the function `getline` instead. (See Section 16.2.6.1 [getline], page 127.)

The `gets` function takes one parameter, the string in which to store the data read. It reads characters from standard input up to the next newline character (that is, when the user presses ⟨RETURN⟩), discards the newline character, and copies the rest into the string passed to it. If there was no error, it returns the same string (as a return value, which may be discarded); otherwise, if there was an error, it returns a null pointer.

Here is a short code example that uses `gets`:

```
#include <stdio.h>

int main()
{
  char my_string[500];
  printf("Type something.\n");
  gets(my_string);
  printf ("You typed: %s\n", my_string);

  return 0;
}
```

If you attempt to compile the example above, it will compile and will run properly, but GCC will warn you against the use of a deprecated function, as follows:

```
/tmp/ccPW3krf.o: In function `main':
/tmp/ccPW3krf.o(.text+0x24): the `gets' function
     is dangerous and should not be used.
```

Remember! Never use this function in your own code. Always use `getline` instead.

16.2.7.2 fgets

The `fgets` ("file get string") function is similar to the `gets` function. This function is *deprecated* — that means it is obsolete and it is strongly suggested you do not use it —

because it is dangerous. It is dangerous because if the input data contains a null character, you can't tell. Don't use `fgets` unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message. Always use `getline` or `getdelim` instead of `fgets` if you can.

Rather than reading a string from standard input, as `gets` does, `fgets` reads it from a specified stream, up to and including a newline character. It stores the string in the string variable passed to it, adding a null character to terminate the string. This function takes three parameters: the first is the string into which to read data, the second is the maximum number of characters to read. (You must supply at least this many characters of space in the string, or your program will probably crash, but at least the `fgets` function protects against overflowing the string and creating a security hazard, unlike `gets`.) The third parameter is the stream from which to read. The number of characters that `fgets` reads is actually one less than than number specified; it stores the null character in the extra character space.

If there is no error, `fgets` returns the string read as a return value, which may be discarded. Otherwise, for example if the stream is already at end of file, it returns a null pointer.

Unfortunately, like the `gets` function, `fgets` is deprecated, in this case because when `fgets` cannot tell whether a null character is included in the string it reads. If a null character is read by `fgets`, it will be stored in the string along with the rest of the characters read. Since a null character terminates a string in C, C will then consider your string to end prematurely, right before the first null character. Only use `fgets` if you are certain the data read cannot contain a null; otherwise, use `getline`.

Here is a code example that uses `fgets`. It will create a text file containing the string '`Hidee ho!`' plus a newline, read it back with `fgets`, and print it on standard output. Notice that although 100 characters are allocated for the string `my_string`, and requested to be read in the `fgets` call, there are not that many characters in the file. The `fgets` function only reads the string up to the newline character; the important thing is to allocate enough space in the string variable to contain the string to be read.

```
#include <stdio.h>

int main()
{
  int input_character;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  char my_string[100];

  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "Hidee ho!\n");

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);
```

```
    my_stream = fopen (my_filename, "r");
    fgets (my_string, 100, my_stream);

    /* Close stream; skip error-checking for brevity of example */
    fclose (my_stream);

    printf ("%s", my_string);

    return 0;
  }
```

16.2.8 Formatted string input

The formatted string input functions are the opposite of the formatted string output functions. Unlike `printf` and similar functions, which generate formatted output, `scanf` and its friends parse formatted input. Like the opposite functions, each accepts, as a parameter, a template string that contains conversion specifiers. In the case of `scanf` and related functions, however, the conversion specifiers are meant to match patterns in an input string, such as integers, floating point numbers, and character sequences, and store the values read in variables.

16.2.8.1 sscanf

The `sscanf` function accepts a string from which to read input, then, in a manner similar to `printf` and related functions, it accepts a template string and a series of related arguments. It tries to match the template string to the string from which it is reading input, using conversion specifier like those of `printf`.

The `sscanf` function is just like the deprecated parent `scanf` function, except that the first argument of `sscanf` specifies a string from which to read, whereas `scanf` can only read from standard input. Reaching the end of the string is treated as an end-of-file condition.

Here is an example of `sscanf` in action:

```
sscanf (input_string, "%as %as %as", &str_arg1, &str_arg2, &str_arg3);
```

If the string `sscanf` is scanning overlaps with any of the arguments, unexpected results will follow, as in the following example. Don't do this!

```
sscanf (input_string, "%as", &input_string);
```

Here is a good code example that parses input from the user with `sscanf`. It prompts the user to enter three integers separated by whitespace, then reads an arbitrarily long line of text from the user with `getline`. It then checks whether exactly three arguments were assigned by `sscanf`. If the line read does not contain the data requested (for example, if it contains a floating-point number or any alphabetic characters), the program prints an error message and prompts the user for three integers again. When the program finally receives exactly the data it was looking for from the user, it prints out a message acknowledging the input, and then prints the three integers.

It is this flexibility of input and great ease of recovery from errors that makes the `getline`/`sscanf` combination so vastly superior to `scanf` alone. Simply put, you should never use `scanf` where you can use this combination instead.

```
#include <stdio.h>

int main()
{
  int nbytes = 100;
  char *my_string;
  int int1, int2, int3;
  int args_assigned;
  args_assigned = 0;

  while (args_assigned != 3)
    {
      puts ("Please enter three integers separated by whitespace.");
      my_string = (char *) malloc (nbytes + 1);
      getline (&my_string, &nbytes, stdin);
      args_assigned = sscanf (my_string, "%d %d %d", &int1, &int2, &int3);
      if (args_assigned != 3)
puts ("\nInput invalid!");
    }
  printf ("\nThanks!\n%d\n%d\n%d\n", int1, int2, int3);

  return 0;
}
```

Template strings for `sscanf` and related functions are somewhat more free-form than those for `printf`. For example, most conversion specifiers ignore any preceding whitespace. Further, you cannot specify a precision for `sscanf` conversion specifiers, as you can for those of `printf`.

Another important difference between `sscanf` and `printf` is that the arguments to `sscanf` must be pointers; this allows `sscanf` to return values in the variables they point to. If you forget to pass pointers to `sscanf`, you may receive some strange errors, and it is easy to forget to do so; therefore, this is one of the first things you should check if code containing a call to `sscanf` begins to go awry.

A `sscanf` template string can contain any number of any number of whitespace characters, any number of ordinary, non-whitespace characters, and any number of conversion specifiers starting with ‘`%`’. A whitespace character in the template string matches zero or more whitespace characters in the input string. Ordinary, non-whitespace characters must correspond exactly in the template string and the input stream; otherwise, a matching error occurs. Thus, the template string ‘`" foo "`’ matches ‘`"foo"`’ and ‘`" foo "`’, but not ‘`" food "`’.

If you create an input conversion specifier with invalid syntax, or if you don’t supply enough arguments for all the conversion specifiers in the template string, your code may do unexpected things, so be careful. Extra arguments, however, are simply ignored.

Conversion specifiers start with a percent sign (‘`%`’) and terminate with a character from the following table:

16.2.8.2 Formatted input conversion specifiers

'c' Matches a fixed number of characters. If you specify a maximum field width (see below), that is how many characters will be matched; otherwise, '%c' matches one character. This conversion does not append a null character to the end of the text it reads, as does the '%s' conversion. It also does not skip whitespace characters, but reads precisely the number of characters it was told to, or generates a matching error if it cannot.

'd' Matches an optionally signed decimal integer, containing the following sequence:

1. An optional plus or minus sign ('+' or '-').
2. One or more decimal digits.

Note that '%d' and '%i' are not synonymous for `scanf`, as they are for `printf`.

'e' Matches an optionally signed floating-point number, containing the following sequence:

1. An optional plus or minus sign ('+' or '-').
2. A floating-point number in decimal or hexadecimal format.
   - The decimal format is a sequence of one or more decimal digits, optionally containing a decimal point character (usually '.'), followed by an optional exponent part, consisting of a character 'e' or 'E', an optional plus or minus sign, and a sequence of decimal digits.
   - The hexadecimal format is a '0x' or '0X', followed by a sequence of one or more hexadecimal digits, optionally containing a decimal point character, followed by an optional binary-exponent part, consisting of a character 'p' or 'P', an optional plus or minus sign, and a sequence of digits.

'E' Same as 'e'.

'f' Same as 'e'.

'g' Same as 'e'.

'G' Same as 'e'.

'i' Matches an optionally signed integer, containing the following sequence:

1. An optional plus or minus sign ('+' or '-').
2. A string of characters representing an unsigned integer.

   If the string begins with '0x' or '0X', the number is assumed to be in hexadecimal format, and the rest of the string must contain hexadecimal digits.

   Otherwise, if the string begins with '0', the number is assumed to be in octal format (base eight), and the rest of the string must contain octal digits.

   Otherwise, the number is assumed to be in decimal format, and the rest of the string must contain decimal digits.

Note that '%d' and '%i' are not synonymous for `scanf`, as they are for `printf`. You can print integers in this syntax with `printf` by using the '#' flag character with the '%x' or '%d' output conversions. (See Section 16.2.2.1 [printf], page 121.)

'`s`' Matches a string of non-whitespace characters. It skips initial whitespace, but stops when it meets more whitespace after it has read something. It stores a null character at the end of the text that it reads, to mark the end of the string. (See Section 16.2.9.2 [String overflows with scanf], page 136, for a warning about using this conversion.)

'`x`' Matches an unsigned integer in hexadecimal format. The string matched must begin with '`0x`' or '`0X`', and the rest of the string must contain hexadecimal digits.

'`X`' Same as '`x`'.

'`[`' Matches a string containing an arbitrary set of characters. For example, '`%12[0123456789]`' means to read a string with a maximum field width of 12, containing characters from the set '`0123456789`' — in other words, twelve decimal digits. An embedded '`-`' character means a range of characters; thus '`%12[0-9]`' means the same thing as the last example. Preceding the characters in the square brackets with a caret ('`^`') means to read a string *not* containing the characters listed. Thus, '`%12[^0-9]`' means to read a twelve-character string not containing any decimal digit. (See Section 16.2.9.2 [String overflows with scanf], page 136, for a warning about using this conversion.)

'`%`' Matches a percent sign. Does not correspond to an argument, and does not permit flags, field width, or type modifier to be specified (see below).

In between the percent sign ('`%`') and the input conversion character, you can place some combination of the following modifiers, in sequence. (Note that the percent sign conversion ('`%%`') doesn't use arguments or modifiers.)

- An optional '`*`' flag. This flag specifies that a match should be made between the conversion specifier and an item in the input stream, but that the value should *not* then be assigned to an argument.
- An optional '`a`' flag, valid with string conversions only. This is a GNU extension to `scanf` that requests allocation of a buffer long enough to safely store the string that was read. (See Section 16.2.9.2 [String overflows with scanf], page 136, for information on how to use this flag.)
- An optional '`'`' flag. This flag specifies that the number read will be grouped according to the rules currently specified on your system. For example, in the United States, this usually means that '`1,000`' will be read as one thousand.
- An optional decimal integer that specifies the maximum field width. The `scanf` function will stop reading characters from the input stream either when this maximum is reached, or when a non-matching character is read, whichever comes first. Discarded initial whitespace does not count toward this width; neither does the null character stored by string input conversions to mark the end of the string.
- An optional type modifier character from the following table. (The default type of the corresponding argument is `int *` for the '`%d`' and '`%i`' conversions, `unsigned int *` for '`%x`' and '`%X`', and `float *` for '`%e`' and its synonyms. You can use these type modifiers to specify otherwise.)

'h' Specifies that the argument to which the value read should be assigned is of type `short int *` or `unsigned short int *`. Valid for the '%d' and '%i' conversions.

'l' For the '%d' and '%i' conversions, specifies that the argument to which the value read should be assigned is of type `long int *` or `unsigned long int *`. For the '%e' conversion and its synonyms, specifies that the argument is of type `double *`.

'L' For the '%d' and '%i' conversions, specifies that the argument to which the value read should be assigned is of type `long long int *` or `unsigned long long int *`. On systems that do not have extra-long integers, this has the same effect as 'l'.

For the '%e' conversion and its synonyms, specifies that the argument is of type `long double *`.

'll' Same as 'L', for the '%d' and '%i' conversions.

'q' Same as 'L', for the '%d' and '%i' conversions.

'z' Specifies that the argument to which the value read should be assigned is of type `size_t`. (The `size_t` type is used to specify the sizes of blocks of memory, and many functions in this chapter use it.) Valid for the '%d' and '%i' conversions.

16.2.9 Deprecated formatted string input functions

These formatted string input functions are generally dangerous and should only be used when there is no alternative. However, because you may encounter them when importing older code or code from non-GNU systems, and because the `scanf` function is in a sense the parent of the safe `sscanf` function, it is important that you know about them.

16.2.9.1 `scanf`

The first of the functions we will examine is `scanf` ("scan formatted"). The `scanf` function is considered dangerous for a number of reasons. First, if used improperly, it can cause your program to crash by reading character strings that overflow the string variables meant to contain them, just like `gets`. (See Section 16.2.7.1 [gets], page 129.) Second, `scanf` can hang if it encounters unexpected non-numeric input while reading a line from standard input. Finally, it is difficult to recover from errors when the `scanf` template string does not match the input exactly.

If you are going to read input from the keyboard, it is far better to read it with `getline` and parse the resulting string with `sscanf` ("string scan formatted") than to use `scanf` directly. However, since `sscanf` uses nearly the same syntax as `sscanf`, as does the related `fscanf`, and since `scanf` is a standard C function, it is important to learn about it.

If `scanf` cannot match the template string to the input string, it will return immediately — and it will leave the first non-matching character as the next character to read from the stream. This is called a *matching error*, and is the main reason `scanf` tends to hang when reading input from the keyboard; a second call to `scanf` will almost certainly choke, since the file position indicator of the stream is not pointing where `scanf` will expect it to. Normally, `scanf` returns the number of assignments made to the arguments it was passed, so check the return value to see if `scanf` found all the items you expected.

16.2.9.2 String overflows with scanf

If you use the '`%s`' and '`%[`' conversions improperly, then the number of characters read is limited only by where the next whitespace character appears. This almost cetainly means that invalid input could make your program crash, because input too long would overflow whatever buffer you have provided for it. No matter how long your buffer is, a user could always supply input that is longer. A well-written program reports invalid input with a comprehensible error message, not with a crash.

Fortunately, it is possible to avoid `scanf` buffer overflow by either specifying a field width or using the '`a`' flag.

When you specify a field width, you need to provide a buffer (using `malloc` or a similar function) of type `char *`. (See Section 20.2 [Memory allocation], page 203, for more information on `malloc`.) You need to make sure that the field width you specify does not exceed the number of bytes allocated to your buffer.

On the other hand, you do not need to allocate a buffer if you specify the '`a`' flag character — `scanf` will do it for you. Simply pass `scanf` an pointer to an unallocated variable of type `char *`, and `scanf` will allocate however large a buffer the string requires, and return the result in your argument. This is a GNU-only extension to `scanf` functionality.

Here is a code example that shows first how to safely read a string of fixed maximum length by allocating a buffer and specifying a field width, then how to safely read a string of any length by using the '`a`' flag.

```
#include <stdio.h>

int main()
{
  int bytes_read;
  int nbytes = 100;
  char *string1, *string2;

  string1 = (char *) malloc (25);

  puts ("Please enter a string of 20 characters or fewer.");
  scanf ("%20s", string1);
  printf ("\nYou typed the following string:\n%s\n\n", string1);

  puts ("Now enter a string of any length.");
  scanf ("%as", &string2);
  printf ("\nYou typed the following string:\n%s\n", string2);

  return 0;
}
```

There are a couple of things to notice about this example program. First, notice that the second argument passed to the first `scanf` call is `string1`, not `&string1`. The `scanf` function requires pointers as the arguments corresponding to its conversions, but a string variable is already a pointer (of type `char *`), so you do not need the extra layer of indirection here. However, you do need it for the second call to `scanf`. We passed it an argument of `&string2` rather than `string2`, because we are using the '`a`' flag, which allocates a string variable big enough to contain the characters it read, then returns a pointer to it.

The second thing to notice is what happens if you type a string of more than 20 characters at the first prompt. The first `scanf` call will only read the first 20 characters, then the second `scanf` call will gobble up all the remaining characters without even waiting for a response to the second prompt. This is because `scanf` does not read a line at a time, the way the `getline` function does. Instead, it immediately matches attempts to match its template string to whatever characters are in the `stdin` stream. The second `scanf` call matches all remaining characters from the overly-long string, stopping at the first whitespace character. Thus, if you type '`12345678901234567890xxxxx`' in response to the first prompt, the program will immediately print the following text without pausing:

```
You typed the following string:
12345678901234567890

Now enter a string of any length.

You typed the following string:
xxxxx
```

(See Section 16.2.8.1 [sscanf], page 131, for a better example of how to parse input from the user.)

16.2.10 fscanf

The `fscanf` function is just like the `scanf` function, except that the first argument of `fscanf` specifies a stream from which to read, whereas `scanf` can only read from standard input.

Here is a code example that generates a text file containing five numbers with `fprintf`, then reads them back in with `fscanf`. Note the use of the '`#`' flags in the '`%#d`' conversions in the `fprintf` call; this is a good way to generate data in a format that `scanf` and related functions can easily read with the '`%i`' input conversion.

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

int main()
{
  float f1, f2;
  int i1, i2;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";

  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "%f %f %#d %#d", 23.5, -12e6, 100, 5);

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);
```

```
  my_stream = fopen (my_filename, "r");
  fscanf (my_stream, "%f %f %i %i", &f1, &f2, &i1, &i2);

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);
  printf ("Float 1 = %f\n", f1);
  printf ("Float 2 = %f\n", f2);
  printf ("Integer 1 = %d\n", i1);
  printf ("Integer 2 = %d\n", i2);

  return 0;
}
```

This code example prints the following output on the screen:

```
Float 1 = 23.500000
Float 2 = -12000000.000000
Integer 1 = 100
Integer 2 = 5
```

If you examine the text file 'snazzyjazz.txt', you will see it contains the following text:

```
23.500000 -12000000.000000 100 5
```

16.3 Single-character input and output

This section covers the use of several functions for the input and output of single characters from standard input and output or files.

16.3.1 getchar

If you want to read a single character from standard input, you can use the **getchar** function. This function takes no parameters, but reads the next character from 'stdin' as an unsigned char, and returns its value, converted to an integer. Here is a short program that uses getchar:

```
#include <stdio.h>

int main()
{
  int input_character;

  printf("Hit any key, then hit RETURN.\n");
  input_character = getchar();
  printf ("The key you hit was '%c'.\n", input_character);
  printf ("Bye!\n");

  return 0;
}
```

Note that because stdin is line-buffered, getchar will not return a value until you hit the ⟨RETURN⟩ key. However, getchar still only reads one character from stdin, so if you type 'hellohellohello' at the prompt, the program above will still only get once character. It will print the following line, and then terminate:

```
The key you hit was 'h'.
Bye!
```

16.3.2 putchar

If you want to print a single character on standard output, you can use the putchar function. It takes a single integer parameter containing a character (the argument can be a single-quoted text character, as in the example below), and sends the character to stdout. If a write error occurs, putchar returns EOF; otherwise, it returns the integer it was passed. This can simply be disregarded, as in the example below.

Here is a short code example that makes use of putchar. It prints an 'X', a space, and then a line of ten exclamation marks ('!!!!!!!!!!') on the screen, then outputs a newline so that the next shell prompt will not occur on the same line. Notice the use of the for loop; by this means, putchar can be used not just for one character, but multiple times.

```
#include <stdio.h>

int main()
{
  int i;

  putchar ('X');
  putchar (' ');
  for (i=1; i<=10; i++)
    {
      putchar ('!');
    }
  putchar ('\n');

  return 0;
}
```

16.3.3 getc and fgetc

If you want to read a single character from a stream other than stdin, you can use the getc function. This function is very similar to getchar, but accepts an argument that specifies the stream from which to read. It reads the next character from the specified stream as an unsigned char, and returns its value, converted to an integer. If a read error occurs or the end of the file is reached, getc returns EOF instead.

Here is a code example that makes use of getc. This code example creates a text file called 'snazzyjazz.txt' with fopen, writes the alphabet in upper-case letters plus a newline to it with fprintf, reads the file position with ftell, and gets the character there with getc. It then seeks position 25 with fseek and repeats the process, attempts to read past the end of the file and reports end-of-file status with feof, and generates an error by attempting to write to a read-only stream. It then reports the error status with ferror, returns to the start of the file with rewind and prints the first character, and finally attempts to close the file and prints a status message indicating whether it could do so.

See Section 16.1.4 [File position], page 116, for information on `ftell`, `fseek`, and `rewind`. See Section 16.1.6 [End-of-file and error functions], page 118, for more information on `feof` and `ferror`.

```
#include <stdio.h>

int main()
{
  int input_char;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  long position;
  int eof_status, error_status, close_error;

  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  printf ("Opening file...\n");
  my_stream = fopen (my_filename, "r");
  position = ftell (my_stream);
  input_char = getc (my_stream);
  printf ("Character at position %d = '%c'.\n\n", position, input_char);

  printf ("Seeking position 25...\n");
  fseek (my_stream, 25, SEEK_SET);
  position = ftell (my_stream);
  input_char = getc (my_stream);
  printf ("Character at position %d = '%c'.\n\n", position, input_char);

  printf ("Attempting to read again...\n");
  input_char = getc (my_stream);
  eof_status = feof (my_stream);
  printf ("feof returns %d.\n\n", eof_status);

  error_status = ferror (my_stream);
  printf ("ferror returns %d.\n", error_status);
  printf ("Attempting to write to this read-only stream...\n");
  putc ('!', my_stream);
  error_status = ferror (my_stream);
  printf ("ferror returns %d.\n\n", error_status);

  printf ("Rewinding...\n");
  rewind (my_stream);
  position = ftell (my_stream);
  input_char = getc (my_stream);
  printf ("Character at position %d = '%c'.\n", position, input_char);
```

```
    close_error = fclose (my_stream);

    /* Handle fclose errors */
    if (close_error != 0)
      {
        printf ("File could not be closed.\n");
      }
    else
      {
        printf ("File closed.\n");
      }

    return 0;
  }
```

There is another function in the GNU C Library called `fgetc`. It is identical to `getc` in most respects, except that `getc` is usually implemented as a macro function and is highly optimised, so is preferable in most situations. (In situations where you are reading from standard input, `getc` is about as fast as `fgetc`, since humans type slowly compared to how fast computers can read their input, but when you are reading from a stream that is not interactively produced by a human, `fgetc` is probably better.)

16.3.4 putc and fputc

If you want to write a single character to a stream other than `stdout`, you can use the `putc` function. This function is very similar to `putchar`, but accepts an argument that specifies the stream to which to write. It takes a single integer parameter containing a character (the argument can be a single-quoted text character, as in the example below), and sends the character to the specified stream. If a write error occurs, `putc` returns `EOF`; otherwise, it returns the integer it was passed. This can simply be disregarded, as in the example below.

The following code example creates a text file called '`snazzyjazz.txt`'. It then writes an '`X`', a space, and then a line of ten exclamation marks ('`!!!!!!!!!!`') to the file, and a newline character to it using the `putc` function. Notice the use of the `for` loop; by this means, `putchar` can be used not just for one character, but multiple times. , then writes ten exclamation mark characters ('`!!!!!!!!!!`')

```
#include <stdio.h>

int main()
{
  int i;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";

  my_stream = fopen (my_filename, "w");
```

```
    putc ('X', my_stream);
    putc (' ', my_stream);
    for (i=1; i<=10; i++)
      {
        putc ('!', my_stream);
      }
    putc ('\n', my_stream);

    /* Close stream; skip error-checking for brevity of example */
    fclose (my_stream);

    return 0;
  }
```

There is another function in the GNU C Library called `fputc`. It is identical to `putc` in most respects, except that `putc` is usually implemented as a macro function and is highly optimised, so is preferable in most situations.

16.3.5 `ungetc()`

Every time a character is read from a stream by a function like `getc`, the file position indicator advances by 1. It is possible to reverse the motion of the file position indicator with the function `ungetc`, which steps the file position indicator back by one byte within the file and reverses the effect of the last character read operation. (This is called *unreading* the character or *pushing it back* onto the stream.)

The intended purpose is to leave the indicator in the correct file position when other functions have moved too far ahead in the stream. Programs can therefore *peek ahead*, or get a glimpse of the input they will read next, then reset the file position with `ungetc`.

On GNU systems, you cannot call `ungetc` twice in a row without reading at least one character in between; in other words, GNU only supports one character of *pushback*.

Pushing back characters does not change the file being accessed at all; `ungetc` only affects the stream buffer, not the file. If `fseek`, `rewind`, or some other file positioning function is called, any character due to be pushed back by `ungetc` is discarded.

Unreading a character on a stream that is at end-of-file resets the end-of-file indicator for the stream, because there is once again a character available to be read. However, if the character pushed back onto the stream is `EOF`, `ungetc` does nothing and just returns `EOF`.

Here is a code example that reads all the whitespace at the beginning of a file with `getc`, then backs up one byte to the first non-whitespace character, and reads all following characters up to a newline character with the `getline` function. (See Section 16.2.6.1 [getline], page 127, for more information on that function.)

```
#include <stdio.h>

int main()
{
  int in_char;
  FILE *my_stream;
  char *my_string = NULL;
  size_t nchars = 0;
```

```
    my_stream = fopen ("snazzyjazz.txt", "w");
    fprintf (my_stream, "          Here's some non-whitespace.\n");

    /* Close stream; skip error-checking for brevity of example */
    fclose (my_stream);
    my_stream = fopen ("snazzyjazz.txt", "r");

    /* Skip all whitespace in stream */
    do
      in_char = getc (my_stream);
    while (isspace (in_char));
    /* Back up to first non-whitespace character */
    ungetc (in_char, my_stream);

    getline (&my_string, &nchars, my_stream);

    /* Close stream; skip error-checking for brevity of example */
    fclose (my_stream);

    printf ("String read:\n");
    printf ("%s", my_string);

    return 0;
  }
```

The code example will skip all initial whitespace in the file '`snazzyjazz.txt`', and display the following text on standard output:

```
String read:
Here's some non-whitespace.
```

16.4 Programming with pipes

There may be times when you will wish to manipulate other programs on a GNU system from within your C program. One good way to do so is the facility called a *pipe*. Using pipes, you can read from or write to any program on a GNU system that writes to standard output and reads from standard input. (In the ancestors of modern GNU systems, pipes were frequently files on disk; now they are usually streams or something similar. They are called "pipes" because people usually visualise data going in at one end and coming out at the other.)

For example, you might wish to send output from your program to a printer. As mentioned in the introduction to this chapter, each printer on your system is assigned a device name such as '`/dev/lp0`'. Pipes provide a better way to send output to the printer than writing directly to the device, however.

Pipes are useful for many things, not just sending output to the printer. Suppose you wish to list all programs and processes running on your computer that contain the string '`init`' in their names. To do so at the GNU/Linux command line, you would type something like the following command:

```
ps -A | grep init
```

This command line takes the output of the `ps -A` command, which lists all running processes, and pipes it with the pipe symbol ('|') to the `grep init` command, which returns all lines that were passed to it that contain the string 'init'. The output of this whole process will probably look something like this on your system:

```
    1 ?        00:00:11 init
 4884 tty6     00:00:00 xinit
```

The pipe symbol '|' is very handy for command-line pipes and pipes within shell scripts, but it is also possible to set up and use pipes within C programs. The two main C functions to remember in this regard are `popen` and `pclose`.

The `popen` function accepts as its first argument a string containing a shell command, such as `lpr`. Its second argument is a string containing either the mode argument 'r' or 'w'. If you specify 'r', the pipe will be open for reading; if you specify 'w', it will be open for writing. The return value is a stream open for reading or writing, as the case may be; if there is an error, `popen` returns a null pointer.

The `pclose` function closes a pipe opened by `popen`. It accepts a single argument, the stream to close. It waits for the stream to close, and returns the status code returned by the program that was called by `popen`.

If you open the pipe for reading or writing, in between the `popen` and `pclose` calls, it is possible to read from or write to the pipe in the same way that you might read from or write to any other stream, with high-level input/output calls such as `getdelim`, `fprintf` and so on.

The following program example shows how to pipe the output of the `ps -A` command to the `grep init` command, exactly as in the GNU/Linux command line example above. The output of this program should be almost exactly the same as sample output shown above.

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

int
main ()
{
  FILE *ps_pipe;
  FILE *grep_pipe;

  int bytes_read;
  int nbytes = 100;
  char *my_string;

  /* Open our two pipes */
  ps_pipe = popen ("ps -A", "r");
  grep_pipe = popen ("grep init", "w");
```

```
  /* Check that pipes are non-null, therefore open */
  if ((!ps_pipe) || (!grep_pipe))
    {
      fprintf (stderr,
               "One or both pipes failed.\n");
      return EXIT_FAILURE;
    }

  /* Read from ps_pipe until two newlines */
  my_string = (char *) malloc (nbytes + 1);
  bytes_read = getdelim (&my_string, &nbytes, "\n\n", ps_pipe);

  /* Close ps_pipe, checking for errors */
  if (pclose (ps_pipe) != 0)
    {
      fprintf (stderr,
               "Could not run 'ps', or other error.\n");
    }

  /* Send output of 'ps -A' to 'grep init', with two newlines */
  fprintf (grep_pipe, "%s\n\n", my_string);

  /* Close grep_pipe, cehcking for errors */
  if (pclose (grep_pipe) != 0)
    {
      fprintf (stderr,
               "Could not run 'grep', or other error.\n");
    }

  /* Exit! */
  return 0;
}
```

16.5 Low-level file routines

High-level file routines such as those already described are usually convenient and easy to use. However, low-level file routines such as the ones in this section have some advantages. For example, they do not treat all file input/output as text streams, as the high-level routines do; for that reason, working with binary files may be easier using low-level routines. For another thing, low-level routines do not buffer their input and output, so you will never need to remember to flush your streams with `fflush` or similar functions, as you sometimes must with high-level routines.

Unfortunately, because low-level routines work at a lower level of abstraction, they can be tricky, even dangerous to use — that is to say, if used incorrectly, they may corrupt your data or cause your program to terminate unexpectedly; never fear, they will not explode your monitor or cause your computer to become sapient and attempt world domination.

As mentioned, low-level file routines do not use text streams; instead, the connection they open to your file is an integer called a *file descriptor*. You pass the file descriptor that designates your file to most low-level file routines, just as you pass the stream that designates your file to most high-level file routines. For example, while the low-level `open` function takes a filename string to open a file, the matched `close` function takes the file descriptor returned by `open`:

```
my_file_descriptor = open ("foo_file", O_RDONLY);
close_err = close (my_file_descriptor);
```

16.5.1 Usual file name errors

Most low-level file functions return some kind of error flag if they cannot perform the action you request, for example, if they cannot parse the file name or find the file. However, to discover *which* error or *what kind of* error has occurred, you must frequently refer to the system variable `errno`. This is an integer specifying the most recent error that has occurred. Macros for values of `errno` are listed below. They are all defined in the GNU C Library.

The word *component* below refers to part of a full file name. For example, in the file name '`/home/fred/snozzberry.txt`', '`fred`' is a component that designates a subdirectory of the directory '`/home`', and '`snozzberry.txt`' is the name of the file proper.

Most functions that accept file name arguments can detect the following error conditions. These are known as the *usual file name errors*. The names of the errors, such as `EACCES`, are compounded of '`E`' for "error" and a term indicating the type of error, such as '`ACCES`' for "access".

`EACCES` The program is not permitted to search within one of the directories in the file name.

`ENAMETOOLONG`
Either the full file name is too long, or some component is too long. GNU does not limit the overall length of file names, but depending on which file system you are using, the length of component names may be limited. (For example, you may be running GNU/Linux but accessing a Macintosh HFS disk; the names of Macintosh files cannot be longer than 31 characters.)

`ENOENT` Either some component of the file name does not exist, or some component is a symbolic link whose target file does not exist.

`ENOTDIR` One of the file name components that is supposed to be a directory is not a directory.

`ELOOP` Too many symbolic links had to be followed to find the file. (GNU has a limit on how many symbolic links can be followed at once, as a basic way to detect recursive (looping) links.)

You can display English text for each of these errors with the '`m`' conversion specifier of the `printf` function, as in the following short example.

```
errno = EACCES;
printf ("errno string (EACCES): %m\n");
```

This example prints the following string:

```
errno string (EACCES): Permission denied
```

See Section 16.2.2.2 [Formatted output conversion specifiers], page 121, for more information on the '`m`' conversion specifier.

16.5.2 Opening files at a low level

You can open a file, or create one if it does not already exist, with the `open` command, which creates and returns a new file descriptor for the file name passed to it. If the file is successfully opened, the file position indicator is initially set to zero (the beginning of the file). (Note that the `open` function is actually called at an underlying level by `fopen`.)

The first parameter of `open` is a string containing the filename of the file you wish to open. The second parameter is an integer argument created by the *bitwise OR* of the following file status flags. (Bitwise OR is a mathematical operator that we have not yet covered in this book. To perform bitwise OR on two variables `a` and `b`, you simply insert a pipe character between them, thus: `a | b`. Bitwise OR is similar to the way the expression "and/or" is used in English. See the code example below for the use of bitwise OR with file status flags. See Chapter 18 [Advanced operators], page 177, for a detailed explanation of bitwise OR and other bitwise operators.)

The following flags are the more important ones for a beginning C programmer to know. There are a number of file status flags which are relevant only to more advanced programmers; for more details, see section "File Status Flags" in *The GNU C Library Reference Manual*.)

Note that these flags are defined in macros in the GNU C Library header file '`fcntl.h`', so remember to insert the line `#include <fcntl.h>` at the beginning of any source code file that uses them.

`O_RDONLY` Open the file for read access.

`O_WRONLY` Open the file for write access.

`O_RDWR` Open the file for both read and write access. Same as `O_RDONLY | O_WRONLY`.

`O_READ`
Same as `O_RDWR`. GNU systems only.

`O_WRITE` Same as `O_WRONLY`. GNU systems only.

`O_EXEC` Open the file for executing. GNU systems only.

`O_CREAT` The file will be created if it doesn't already exist.

`O_EXCL` If `O_CREAT` is set as well, then `open` fails if the specified file exists already. Set this flag if you want to ensure you will not clobber an existing file.

`O_TRUNC` Truncate the file to a length of zero bytes. This option is not useful for directories or other such special files. You must have write permission for the file, but you do not need to open it for write access to truncate it (under GNU).

`O_APPEND` Open the file for appending. All `write` operations then write the data at the end of the file. This is the only way to ensure that the data you write will always go to the end of the file, even if there are other `write` operations happening at the same time.

The `open` function normally returns a non-negative integer file descriptor connected to the specified file. If there is an error, `open` will return -1 instead. In that case, you can check the `errno` variable to see which error occurred. In addition to the usual file name errors, `open` can set `errno` to the following values. (It can also specify a few other errors of interest only to advanced C programmers. See section "Opening and Closing Files" in *The GNU C Library Reference Manual*, for a full list of error values. See Section 16.5.1 [Usual file name errors], page 146, for a list of the usual file name errors.).

`EACCES` The file exists but is cannot be does not have read or write access (as requested), or the file does not exist but cannot be created because the directory does not have write access.

`EEXIST` Both `O_CREAT` and `O_EXCL` are set, and the named file already exists. To open it would clobber it, so it will not be opened.

`EISDIR` Write access to the file was requested, but the file is actually a directory.

`EMFILE` Your program has too many files open.

`ENOENT` The file named does not exist, and `O_CREAT` was not specified, so the file will not be created.

`ENOSPC` The file cannot be created, because the disk is out of space.

`EROFS` The file is on a read-only file system, but either one of `O_WRONLY`, `O_RDWR`, or `O_TRUNC` was specified, or `O_CREAT` was set and the file does not exist.

See Section 16.5.3 [Closing files at a low level], page 148, for a code example using both the low-level file functions `open` and `close`.

16.5.2.1 File creation

In older C code using low-level file routines, there was a function called `creat` that was used for creating files. This function is still included in GNU for compatibility with older C code, but is considered obsolete. In order to create a file, instead of writing

```
creat (filename)
```

it is now considered better coding to practice to write the following code:

```
open (filename, O_WRONLY | O_CREAT | O_TRUNC)
```

16.5.3 Closing files at a low level

To close a file descriptor, use the low-level file function `close`. The sole argument to `close` is the file descriptor you wish to close.

The `close` function returns 0 if the call was successful, and -1 if there was an error. In addition to the usual file name error codes, it can set the system variable `errno` to one of the following values. It can also set `errno` to several other values, mostly of interest to advanced C programmers. See section "Opening and Closing Files" in *The GNU C Library Reference Manual*, for more information.

`EBADF` The file descriptor passed to `close` is not valid.

Remember, close a stream by using `fclose` instead. This allows the necessary system bookkeeping to take place before the file is closed.

Here is a code example using both the low-level file functions `open` and `close`.

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

int main()
{

  char my_filename[] = "snazzyjazz17.txt";
  int my_file_descriptor, close_err;

  /*
    Open my_filename for writing.  Create it if it does not exist.
    Do not clobber it if it does.
  */

  my_file_descriptor = open (my_filename, O_WRONLY | O_CREAT | O_EXCL);
  if (my_file_descriptor == -1)
    {
      printf ("Open failed.\n");
    }

  close_err = close (my_file_descriptor);
  if (close_err == -1)
    {
      printf ("Close failed.\n");
    }

  return 0;
}
```

Running the above code example for the first time should produce no errors, and should create an empty text file called 'snazzyjazz17.txt'. Running it a second time should display the following errors on your monitor, since the file 'snazzyjazz17.txt' already exists, and should not be clobbered according to the flags passed to open.

```
Open failed.
Close failed.
```

16.5.4 Reading files at a low level

You can read a block of information from a file with the read function. The data read is loaded directly into a buffer in memory. The data can be binary as well as a text, but if the latter, no terminating newline is added. The bytes read start at the current file position; after reading them, read advances the file position to immediately after the bytes read.

The read function takes three parameters. The first one is the file descriptor from which data is to be read. The second is the buffer in memory where the data read will be stored. The buffer is of type void *, and can be an array or a chunk of space reserved with malloc. The final parameter is of type size_t, and specifies the number of bytes to read.

The return value of this function is of type ssize_t, and represents the number of bytes actually read. This might be less than the number of bytes requested if there are not enough bytes left in the file or immediately available. Reading less than the number of bytes requested does not generate an error.

If the number of bytes requested is not zero, a return value of zero indicates the end of the file. This is also not an error. If you keep calling `read` at the end of the file, it will simply keep returning zero. If `read` returns at least one character, you cannot tell whether the end of the file was reached from that information, but `read` will return zero on the next read operation if it was.

If there was an error, `read` returns -1. You can then check the system variable `errno` for one of the following error conditions, as well as the usual file name errors. (See Section 16.5.1 [Usual file name errors], page 146.) The `read` function can also return some other error conditions in `errno` that are mostly of interest to advanced C programmers. (See section "Input and Output Primitives" in *The GNU C Library Reference Manual*, for more information.)

`EBADF` The file descriptor passed to `read` is not valid, or is not open for reading.

`EIO` There was a hardware error. (This error code also applies to more abstruse conditions detailed in the GNU C Library manual.)

See Section 16.5.5 [Writing files at a low level], page 150, for a code example that uses the `read` function.

16.5.5 Writing files at a low level

You can write a block of information to a file with the `write` function, which is called by all high-level file writing routines, such as `fwrite`. It takes three parameters. The first is the file descriptor of the file you wish to write to. The second is a buffer, of type `void *`, that contains the data you wish to write. (It can be an array of bytes, but need not be a text string. Null characters in the data are treated in the same way as other characters.) The third parameter is of type `size_t`, and specifies the number of bytes that are to be written.

The return value of this function is of type `ssize_t`, and indicates the number of bytes actually written. This may be the same as the third parameter (the number of bytes to be written), but may be less; you should always call `write` in a loop, and iterate the loop until all data has been written. If there is an error, `write` returns -1. The `write` function will return the following error codes in the system variable `errno`, as well as the usual file name errors. (See Section 16.5.1 [Usual file name errors], page 146.)

`EBADF` The file descriptor specified is invalid, or is not open for writing.

`EFBIG` If the data were written, the file written to would become too large.

`EIO` There has been a hardware error.

`EINTR` The write operation was temporarily interrupted.

`ENOSPC` The device containing the file is full.

In addition to the error codes above, `write` can return some error codes that are mainly of interest to advanced C programmers. If `write` fails, you should check `errno` to see if the error was `EINTR`; if it was, you should repeat the `write` call each time.

Even though low-level file routines do not use buffering, and once you call `write`, your data can be read from the file immediately, it may take up to a minute before your data is physically written to disk. You can call the `fsync` routine (see below) to ensure that all data is written to the file; this usage is roughly analogous to the high-level file routine `fflush`.

The `fsync` routine takes a single parameter, the file descriptor to synchronise. It does not return a value until all data has been written. If no error occurred, it returns a 0; otherwise, it returns -1 and sets the system variable `errno` to one of the following values:

`EBADF` The file descriptor specified is invalid.

`EINVAL` No synchronization is possible because the system does not implement it.

Here is a code example that demonstrates the use of the `write`, `read`, and `fsync` functions. (See Section 16.5.4 [Reading files at a low level], page 149, for more information on `read`.)

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

int main()
{
  char my_write_str[] = "1234567890";
  char my_read_str[100];
  char my_filename[] = "snazzyjazz.txt";
  int my_file_descriptor, close_err;

  /* Open the file.  Clobber it if it exists. */
  my_file_descriptor = open (my_filename, O_RDWR | O_CREAT | O_TRUNC);

  /* Write 10 bytes of data and make sure it's written */
  write (my_file_descriptor, (void *) my_write_str, 10);
  fsync (my_file_descriptor);

  /* Seek the beginning of the file */
  lseek (my_file_descriptor, 0, SEEK_SET);

  /* Read 10 bytes of data */
  read (my_file_descriptor, (void *) my_read_str, 10);

  /* Terminate the data we've read with a null character */
  my_read_str[10] = '\0';

  printf ("String read = %s.\n", my_read_str);

  close (my_file_descriptor);

  return 0;
}
```

16.5.6 Finding file positions at a low level

If you want to find a particular file position within a file, using a low-level file routine, you can call the `lseek` function. This is very similar to the high-level file routine `fseek`, except that it accepts a file descriptor rather than a stream as an argument.

The `lseek` function specifies the file position for the next `read` or `write` operation. (See Section 16.1.4 [File position], page 116, for more information on file positions.)

The `lseek` function takes three parameters. The first parameter is the file descriptor. The second is of type `off_t` and specifies the number of bytes to move the file position indicator. The third argument and the third parameter is a constant that specifies whether the offset is relative to the beginning of the file (`SEEK_SET`), to the current file position (`SEEK_CUR`), or to the end of the file (`SEEK_END`). If `SEEK_CUR` or `SEEK_END` is used, the offset specified can be positive or negative. If you specify `SEEK_END`, set the position past the current end of the file, and actually write data, you will extend the file with zeroes up to the position you specify. However, the blocks of zeroes are not actually written to disk, so the file takes up less space on disk than it seems to; this is called a *sparse file*.

The return value of `lseek` is of type `off_t` and normally contains the resulting file position, as measured in bytes from the beginning of the file. If you wish to read the current file position, therefore, you can specify an offset of 0 and a third parameter of `SEEK_CUR`, as follows:

file_position `= lseek (`*file_descriptor*`, 0, SEEK_CUR);`

If there was an error, `lseek` returns a -1 and sets the system variable `errno` to one of the following values:

`EBADF` The file descriptor specified is invalid.

`EINVAL` Either the third parameter of `lseek` is invalid, or the file offset is invalid.

`ESPIPE` The file descriptor corresponds to an object that cannot be positioned, such as a terminal device.

The `lseek` function is called by many high-level file position functions, including `fseek`, `rewind`, and `ftell`.

16.5.7 Deleting files at a low level

If you want to delete a file, you can use the low-level file routine `unlink`, as declared in the file '`unistd.h`'. Simply pass this routine the name of the file you wish to delete. If this is the only name the file has (that is, if no one has created a hard link to the file with the `link` function, the GNU command `ln`, or something similar), then the file itself will be deleted; otherwise, only that name will be deleted. (See the section "Hard Links" in the GNU C Library manual for more information on hard links.) If the file is open when `unlink` is called, `unlink` will wait for the file to be closed before it deletes it.

The `unlink` function returns 0 if the file or file name was successfully deleted. If there was an error, `unlink` returns -1. In addition to the usual file name errors, `unlink` can set `errno` to the following values. (See Section 16.5.1 [Usual file name errors], page 146, for a list of the usual file name errors.)

`EACCES` Your program does not have permission to delete the file from the directory that contains it.

`EBUSY` The file is currently being used by the system and cannot be deleted.

`ENOENT` The file name to be deleted does not exist.

`EPERM` Your program tried to delete a directory with `unlink`; this is not permitted under GNU. (See `remove` below.)

`EROFS` The file name is on a read-only file system and cannot be deleted.

If you wish to delete a directory rather than an ordinary file, use the `rmdir` function. Simply pass it the name of an empty directory you wish to delete. It acts like `unlink` in most respects, except that it can return an extra error code in the system variable `errno`:

`ENOTEMPTY`
The directory was not empty, so cannot be deleted. This code is synonymous with `EEXIST`, but GNU always returns `ENOTEMPTY`.

16.5.8 Renaming files at a low level

If you want to rename a file, you can use the `rename` function, which takes two parameters. The first parameter is a string containing the old name of the file, and the second is a string containing the new name. (As with `unlink`, this function only operates on one of the names of a file, if the file has hard links. See Section 16.5.7 [Deleting files at a low level], page 152, for caveats and information on hard links.)

Both the new name and the old name must be on the same file system. Any file in the same directory that has the same name as the new file name will be deleted in the process of renaming the file.

If `rename` fails, it will return -1. In addition to the usual file name errors, `unlink` can set `errno` to the following values. (See Section 16.5.1 [Usual file name errors], page 146, for a list of the usual file name errors.)

`EACCES` Either one of the directories in question (either the one containing the old name or the one containing the new name) refuses write permission, or the new name and the old name are directories, and write permission is refused for at least one of them.

`EBUSY` One of the directories used by the old name or the new name is being used by the system and cannot be changed.

`ENOTEMPTY`
The directory was not empty, so cannot be deleted. This code is synonymous with `EEXIST`, but GNU always returns `ENOTEMPTY`.

`EINVAL` The old name is a directory that contains the new name.

`EISDIR` The new name is a directory, but the old name is not.

`EMLINK` The parent directory of the new name would contain too many entries if the new name were created.

`ENOENT` The old name does not exist.

`ENOSPC` The directory that would contain the new name has no room for another entry, and cannot be expanded.

`EROFS` The rename operation would involve writing on a read-only file system.

`EXDEV` The new name and the old name are on different file systems.

16.6 Questions

1. What are the following?
   1. File name
   2. File descriptor
   3. Stream
2. What is a pseudo-device name?
3. Where does '`stdin`' usually get its input?
4. Where does '`stdout`' usually send its output?
5. Write a program that simply prints out the following string to the screen: '`6.23e+00`'.
6. Investigate what happens when you type the wrong conversion specifier in a program. e.g. try printing an integer with '`%f`' or a floating point number with '`%c`'. This is bound to go wrong – but how will it go wrong?
7. What is wrong with the following statements?
   1. `printf (x);`
   2. `printf ("%d");`
   3. `printf ();`
   4. `printf ("Number = %d");`

   Hint: if you don't know, try them in a program!
8. What is a whitespace character?
9. Write a program that aceepts two integers from the user, multiplies them together, and prints the answer on your printer. Try to make the input as safe as possible.
10. Write a program that simply echoes all the input to the output.
11. Write a program that strips all space characters out of the input and replaces each string of them with a single newline character.
12. The `scanf` function always takes pointer arguments. True or false?
13. What is the basic difference between high-level and low-level file routines?
14. Write a statement that opens a high level file for reading.
15. Write a statement that opens a low level file for writing.
16. Write a program that checks for illegal characters in text files. The only valid characters are ASCII codes 10, 13, and 32..126.
17. What statement performs formatted writing to text files?
18. Poke around in the header files on your system so you can see what is defined where.

17 Putting a program together

This chapter explains, step by step, how to create a "real" program that meets GNU standards for a command-line interface. It also discusses how to create a program whose source is split into multiple files, and how to compile it, with or without the GNU utility make. Finally, it discusses how to create a code library, in case you write some useful functions that you want to share with other programmers.

Putting it all together.

17.1 argc and argv

So far, all the programs we have written can be run with a single command. For example, if we compile an executable called 'myprog', we can run it from within the same directory with the following command at the GNU/Linux command line:

```
./myprog
```

However, what if you want to pass information from the command line to the program you are running? Consider a more complex program like GCC. To compile the hypothetical 'myprog' executable, we type something like the following at the command line:

```
gcc -o myprog myprog.c
```

The character strings '-o', 'myprog', and 'myprog.c' are all *arguments* to the gcc command. (Technically 'gcc' is an argument as well, as we shall see.)

Command-line arguments are very useful. After all, C functions wouldn't be very useful if you couldn't ever pass arguments to them — adding the ability to pass arguments to programs makes them that much more useful. In fact, all the arguments you pass on the command line end up as arguments to the main function in your program.

Up until now, the skeletons we have used for our C programs have looked something like this:

```
#include <stdio.h>

int main()
{

  return 0;
}
```

From now on, our examples may look a bit more like this:

```
#include <stdio.h>

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

  return 0;
}
```

As you can see, main now has arguments. The name of the variable argc stands for "argument count"; argc contains the number of arguments passed to the program. The name of the variable argv stands for "argument vector". A vector is a one-dimensional array, and `argv` is a one-dimensional array of strings. Each string is one of the arguments that was passed to the program.

For example, the command line

```
gcc -o myprog myprog.c
```

would result in the following values internal to GCC:

`argc` 4

`argv[0]` ‘gcc’

`argv[1]` ‘-o’

`argv[2]` ‘myprog’

`argv[3]` ‘myprog.c’

As you can see, the first argument (`argv[0]`) is the name by which the program was called, in this case ‘gcc’. Thus, there will always be at least one argument to a program, and `argc` will always be at least 1.

The following program accepts any number of command-line arguments and prints them out:

```
#include <stdio.h>

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

  printf ("This program was called with \"%s\".\n",argv[0]);
  if (argc > 1)
    {
      for (count = 1; count < argc; count++)
{
  printf("argv[%d] = %s\n", count, argv[count]);
}
    }
  else
    {
      printf("The command had no other arguments.\n");
    }

  return 0;
}
```

If you name your executable ‘fubar’, and call it with the command ‘./fubar a b c’, it will print out the following text:

```
This program was called with "./fubar".
argv[1] = a
argv[2] = b
argv[3] = c
```

17.2 Processing command-line options

It is easy, though tedious to pull options directly out of the `argv` vector with your own routines. It is slightly less tedious to use the standard C option-processing function `getopt`, or the enhanced GNU version of the same function, `getopt_long`, which permits GNU-style *long options* (for example, '`--quiet`' as opposed to '`-q`').

The best option of all is to use the `argp` interface for processing options. Professionally written programs provide the user with standard and useful options. The `argp` function provides for these. For the modest price of setting up your command line arguments in a structured way, and with surprisingly few lines of code, you can obtain all the perks of a "real" GNU program, such as "automagically"-generated output to the '`--help`', '`--usage`', and '`--version`' options, as defined by the GNU coding standards. Using `argp` results in a more consistent look-and-feel for programs that use it, and makes it less likely that the built-in documentation for a program will be wrong or out of date.

POSIX, the Portable Operating System Interface standard, recommends the following conventions for command-line arguments. The `argp` interface makes implementing them easy.

- Command-line arguments are *options* if they begin with a hyphen ('`-`').
- Multiple options may follow a hyphen in a cluster if they do not take arguments. Thus, '`-abc`' and '`-a -b -c`' are the same.
- Option names are single alphanumeric characters.
- Options may require an argument. For example, the '`-o`' option of the `ld` command requires an output file name.
- The whitespace separating an option and its argument is optional. Thus, '`-o foo`' and '`-ofoo`' are the same.
- Options usually precede non-option arguments. (In fact, `argp` is more flexible than this; if you want to suppress this flexibility, define the `_POSIX_OPTION_ORDER` environment variable.)
- The argument '`--`' terminates all options; all following command-line arguments are considered non-option arguments, even if they begin with a hyphen.
- A single hyphen as an argument is considered a non-option argument; by convention, it is used to specify input from standard input or output to standard output.
- Options may appear in any order, even multiple times. The meaning of this is left to the application.

In addition, GNU adds *long options*, like the '`--help`', '`--usage`', and '`--version`' options mentioned above. A long option starts with '`--`', which is then followed by a string of alphanumeric characters and hyphens. Option names are usually one to three words long, with hyphens to separate words. Users can abbreviate the option names as long as the abbreviations are unique. A long option (such as '`--verbose`') often has a short-option synonym (such as '`-v`').

Long options can accept optional (that is, non-necessary) arguments. You can specify an argument for a long option as follows:

'`--`'*option-name*'`=`'*value*

You may not type whitespace between the option name and the equals sign, or between the equals sign and the option value.

17.2.1 `argp` description

This section will describe how to write a simple program that implements most of the standards mentioned above. It assumes some knowledge of advanced C data structures that we have not yet covered in this book; if you are confused, you might want to consult the chapter that discusses this material. (See Chapter 19 [More data types], page 189.) Note that we are only discussing the basics of `argp` in this chapter; to read more about this complicated and flexible facility of the GNU C Library, consult section "Parsing Program Options with Argp" in *The GNU C Library Reference Manual*. Nevertheless, what you learn in this chapter may be all you need to develop a program that is compliant with GNU coding standards, with respect to command-line options.

The main interface to `argp` is the `argp_parse` function. Usually, the only argument-parsing code you will need in `main` is a call to this function. The first parameter it takes is of type `const struct argp *argp`, and specifies an `ARGP` structure (see below). (A value of zero is the same as a structure containing all zeros.) The second parameter is simply `argc`, the third simply `argv`. The fourth parameter is a set of flags that modify the parsing behaviour; setting this to zero usually doesn't hurt unless you're doing something fancy, and the same goes for the fifth parameter. The sixth parameter can be useful; in the example below, we use it to pass information from `main` to our function `parse_opt`, which does most of the work of initalizing internal variables (fields in the `arguments` structure) based on command-line options and arguments.

The `argp_parse` returns a value of type `error_t`: usually either 0 for success, `ENOMEM` if a memory allocation error occurred, or `EINVAL` if an unknown option or argument was met with.

For this example, we are using only the first four fields in `ARGP`, which are usually all that is needed. The rest of the fields will default to zero. The four fields are, in order:

1. `OPTIONS`: A pointer to a vector the elements of which are of type `struct argp_option`, which contains four fields. The vector elements specify which options this parser understands. If you assign your option structure by initializing the array as we do in this section's main example, unspecified fields will default to zero, and need not be specified. The whole vector may contain zero if there are no options at all. It should in any case be terminated by an entry with a zero in all fields (as we do by specifying the last item in the `options` vector to be `{0}` in the main example below.

   The four main `argp_option` structure fields are as follows. (We will ignore the fifth one, which is relatively unimportant and will simply default to zero if you do not specify it.)

   1. `NAME`: The name of this option's long option (may be zero). To specify multiple names for an option, follow it with additional entries, with the `OPTION_ALIAS` flag set.
   2. `KEY`: The integer key to pass to the `PARSER` function when parsing the current option; this is the same as the name of the current option's short option, if it is a printable ASCII character.

3. ARG: The name of this option's argument, if any.
4. FLAGS: Flags describing this option. You can specify multiple flags with logical OR (for example, OPTION_ARG_OPTIONAL | OPTION_ALIAS).

   Some of the available options are:

   - OPTION_ARG_OPTIONAL: The argument to the current option is optional.
   - OPTION_ALIAS: The current option is an alias for the previous option.
   - OPTION_HIDDEN: Don't show the current option in --help output.
5. DOC: A documentation string for the current option; will be shown in --help output.

2. PARSER: A pointer to a function to be called by argp for each option parsed. It should return one of the following values:

   - 0: Success.
   - ARGP_ERR_UNKNOWN: The given key was not recognized.
   - An errno value indicating some other error. (See Section 16.5.1 [Usual file name errors], page 146.)

   The parser function takes the following arguments:

   1. KEY: An integer specifying which argument this is, taken from the KEY field in each argp_option structure, or else a key with a special meaning, such as one of the following:
      - ARGP_KEY_ARG: The current command-line argument is not an option.
      - ARGP_KEY_END: All command-line arguments have been parsed.
   2. ARG: The string value of the current command-line argument, or NULL if it has none.
   3. STATE: A pointer to an argp_state structure, containing information about the parsing state, such as the following fields:
      1. input: The same as the last parameter to argp_parse. We use this in the main code example below to pass information between the main and parse_opt functions.
      2. arg_num: The number of the current non-option argument being parsed.

3. ARGS_DOC: If non-zero, a string describing how the non-option arguments should look. It is only used to print the 'Usage:' message. If it contains newlines, the strings separated by them are considered alternative usage patterns, and printed on separate lines (subsequent lines are preceded by 'or:' rather than 'Usage:'.
4. DOC: If non-zero, a descriptive string about this program. It will normally be printed before the options in a help message, but if you include a vertical tab character ('\v'), the part after the vertical tab will be printed following the options in the output to the --help option. Conventionally, the part before the options is just a short string that says what the program does, while the part afterwards is longer and describes the program in more detail.

There are also some utility functions associated with argp, such as argp_usage, which prints out the standard usage message. We use this function in the parse_opt function in the following example. See section "Functions For Use in Argp Parsers" in *The GNU C Library Reference Manual*, for more of these utility functions.

17.2.2 argp example

Here is a code example that uses argp to parse command-line options. Remember, to compile this example, copy it to a file called something like 'argex.c', then compile it with the command gcc -o argex argex.c and run the resulting binary with the command ./argex.

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

const char *argp_program_version =
"argex 1.0";

const char *argp_program_bug_address =
"<bug-gnu-utils@gnu.org>";

/* This structure is used by main to communicate with parse_opt. */
struct arguments
{
  char *args[2];            /* ARG1 and ARG2 */
  int verbose;              /* The -v flag */
  char *outfile;            /* Argument for -o */
  char *string1, *string2;  /* Arguments for -a and -b */
};

/*
   OPTIONS.  Field 1 in ARGP.
   Order of fields: {NAME, KEY, ARG, FLAGS, DOC}.
*/
static struct argp_option options[] =
{
  {"verbose", 'v', 0, 0, "Produce verbose output"},
  {"alpha",   'a', "STRING1", 0,
   "Do something with STRING1 related to the letter A"},
  {"bravo",   'b', "STRING2", 0,
   "Do something with STRING2 related to the letter B"},
  {"output",  'o', "OUTFILE", 0,
   "Output to OUTFILE instead of to standard output"},
  {0}
};
```

```
/*
   PARSER. Field 2 in ARGP.
   Order of parameters: KEY, ARG, STATE.
*/
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'v':
      arguments->verbose = 1;
      break;
    case 'a':
      arguments->string1 = arg;
      break;
    case 'b':
      arguments->string2 = arg;
      break;
    case 'o':
      arguments->outfile = arg;
      break;
    case ARGP_KEY_ARG:
      if (state->arg_num >= 2)
{
  argp_usage(state);
}
      arguments->args[state->arg_num] = arg;
      break;
    case ARGP_KEY_END:
      if (state->arg_num < 2)
{
  argp_usage (state);
}
      break;
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/*
   ARGS_DOC. Field 3 in ARGP.
   A description of the non-option command-line arguments
     that we accept.
*/
static char args_doc[] = "ARG1 ARG2";
```

```
/*
  DOC.  Field 4 in ARGP.
  Program documentation.
*/
static char doc[] =
"argex -- A program to demonstrate how to code command-line options
and arguments.\vFrom the GNU C Tutorial.";

/*
   The ARGP structure itself.
*/
static struct argp argp = {options, parse_opt, args_doc, doc};

/*
   The main function.
   Notice how now the only function call needed to process
   all command-line options and arguments nicely
   is argp_parse.
*/
int main (int argc, char **argv)
{
  struct arguments arguments;
  FILE *outstream;

  char waters[] =
"a place to stay
enough to eat
somewhere old heroes shuffle safely down the street
where you can speak out loud
about your doubts and fears
and what's more no-one ever disappears
you never hear their standard issue kicking in your door
you can relax on both sides of the tracks
and maniacs don't blow holes in bandsmen by remote control
and everyone has recourse to the law
and no-one kills the children anymore
and no-one kills the children anymore
  --\"the gunners dream\", Roger Waters, 1983\n";

  /* Set argument defaults */
  arguments.outfile = NULL;
  arguments.string1 = "";
  arguments.string2 = "";
  arguments.verbose = 0;
```

```
  /* Where the magic happens */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  /* Where do we send output? */
  if (arguments.outfile)
    outstream = fopen (arguments.outfile, "w");
  else
    outstream = stdout;

  /* Print argument values */
  fprintf (outstream, "alpha = %s\nbravo = %s\n\n",
   arguments.string1, arguments.string2);
  fprintf (outstream, "ARG1 = %s\nARG2 = %s\n\n",
   arguments.args[0],
   arguments.args[1]);

  /* If in verbose mode, print song stanza */
  if (arguments.verbose)
    fprintf (outstream, waters);

  return 0;
}
```

Compile the code, then experiment! For example, here is the program output if you simply type `argex`:

```
Usage: argex [OPTION...] ARG1 ARG2
Try `argex --help' or `argex --usage' for more information.
```

Here is the output from `argex --usage`:

```
Usage: argex [-v?V] [-a STRING1] [-b STRING2] [-o OUTFILE] [--alpha=STRING1]
            [--bravo=STRING2] [--output=OUTFILE] [--verbose] [--help] [--usage]
            [--version] ARG1 ARG2
```

Here is the output from `argex --help`:

```
Usage: argex [OPTION...] ARG1 ARG2
argex -- A program to demonstrate how to code command-line options
and arguments.

  -a, --alpha=STRING1        Do something with STRING1 related to the letter A
  -b, --bravo=STRING2        Do something with STRING2 related to the letter B
  -o, --output=OUTFILE       Output to OUTFILE instead of to standard output
  -v, --verbose              Produce verbose output
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

From the GNU C Tutorial.

Report bugs to <bug-gnu-utils@gnu.org>.
```

Here is the output from `argex Foo Bar`:

```
alpha =
bravo =

ARG1 = Foo
ARG2 = Bar
```

And finally, here is the output from `argex --verbose -a 123 --bravo=456 Foo Bar`:

```
alpha = 123
bravo = 456

ARG1 = Foo
ARG2 = Bar

a place to stay
enough to eat
somewhere old heroes shuffle safely down the street
where you can speak out loud
about your doubts and fears
and what's more no-one ever disappears
you never hear their standard issue kicking in your door
you can relax on both sides of the tracks
and maniacs don't blow holes in bandsmen by remote control
and everyone has recourse to the law
and no-one kills the children anymore
and no-one kills the children anymore
  --"the gunners dream", Roger Waters, 1983
```

You can of course also send the output to a text file with the '`-o`' or '`--output`' option.

17.3 Environment variables

Sometimes it is useful to communicate with a program in a semi-permanent way, so that you do not need to specify a command-line option every time you type the command to execute the program. One way to do this is to generate a configuration file, in which you can store data that will be used by the program every time it is run. This approach is typically useful if you have a large amount of data that you want to pass to a program every time it runs, or if you want the program itself to be able to change the data.

However, *environment variables* provide a more lightweight approach. Environment variables, sometimes called *shell variables*, are usually set with the `export` command in the shell. (This section assumes you are using the GNU Bash shell.) Standard environment variables are used for information about your home directory, terminal type, and so on; you can define additional variables for other purposes. The set of all environment variables that have values is collectively known as the *environment*.

Names of environment variables are case-sensitive, and it is good form to use all upper-case letters when defining a new variable; certainly this is the case for all system-defined environment variables.

The value of an environment variable can be any string that does not contain a null character (since the null character is used to terminate the string value).

Environment variables are stored in a special array that can be read by your `main` function. Here is the skeleton for a `main` function that can read environment variables; notice we have added a third parameter to `main`, called `envp`, which comes after `argc` and `argv`.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{

  return 0;
}
```

Notice that `envp` is an array of strings, just as `argv` is. It consists of a list of the environment variables of your shell, in the following format:

NAME=*value*

Just as you can manually process command-line options from `argv`, so can you manually process environment variables from `envp`. However, the simplest way to access the value of an environment variable is with the `getenv` function, defined in the system header '`stdlib.h`'. It takes a single argument, a string containing the name of the variable whose value you wish to discover. It returns that value, or a null pointer if the variable is not defined.

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

/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{
  char *home, *host;

  home = getenv("HOME");
  host = getenv("HOSTNAME");

  printf ("Your home directory is %s on %s.\n", home, host);

  return 0;
}
```

When you run this code, it will print out a line like the following one.

```
Your home directory is /home/rwhe on linnaeus.
```

Note: Do not modify strings returned from `getenv`; they are pointers to data that belongs to the system. If you want to process a value returned from `getenv`, copy it to another string first with `strcpy`. (See Chapter 15 [Strings], page 101.) If you want to change an environment variable from within your program (not usually advisable), use the `putenv`, `setenv`, and `unsetenv` functions. See section "Environment Access" in *The GNU C Library Reference Manual*, for more information on these functions.

17.4 Compiling multiple files

It is usually very simple to compile a program that has been divided across multiple source files. Instead of typing

```
gcc -o executable sourcefile.c
```

you would type

```
gcc -o executable sourcefile_1.c sourcefile_2.c ... sourcefile_n.c
```

For example, if you were building a simple database program called '`mydb`', the command line might look something like this:

```
gcc -o mydb main.c keyboard_io.c db_access.c sorting.c
```

Of course, if (say) '`db_access.c`' were lengthy, it might take a long time to compile your program every time you executed this command, even if you only made a small change in one of the other files. To avoid this, you might want to compile each of the source files into its own object file, then link them together to make your program. If you did, each time you made a small change in one file, you need only recompile that single file and then link the object files together again, potentially a great savings in time and patience. Here is how to generate a permanent object file for '`db_access.c`'.

```
gcc -c db_access.c
```

This would generate a permanent object code file called '`db_access.o`', indicated by the suffix '`.o`'. You would perform this step when needed for each of the source code files, then link them together with the following command line:

```
gcc -o mydb main.o keyboard_io.o db_access.o sorting.o
```

You might even put the various commands into a shell file, so that you wouldn't need to type them repeatedly. For example, you could put the last command line into a shell file called '`build`', so that all you would need to do to build your executable from object code files is type the following line.

```
./build
```

For programs on a very small scale, this approach works quite well. If your project grows even slightly complex, however, you will have a hard time keeping track of which object files are "fresh" and which need to be recreated because the corresponding source files have been changed since their last compilation. That's where the GNU utility `make` comes in. (See Section 17.5 [Writing a makefile], page 166.)

17.5 Writing a makefile

The GNU `make` program automatically determines which pieces of a large program need to be recompiled, and issues the commands to compile them. You need a file called a *makefile* to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

In this section, we will discuss a simple makefile that describes how to compile and link a text editor which consists of eight C source files and three header files. The makefile can also tell `make` how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation).

Although the examples in this section show C programs, you can use `make` with any programming language whose compiler can be run with a shell command. Indeed, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

Your makefile describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The `make` program uses the makefile database and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the database.

You can provide command line arguments to `make` to control which files should be recompiled, or how.

When `make` recompiles the editor, each changed C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

17.5.1 What a Rule Looks Like

A simple makefile consists of "rules" with the following shape:

target ... : *prerequisites* ...
        command
        ...
        ...

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as '`clean`'.

A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

A *command* is an action that `make` carries out. A rule may have more than one command, each on its own line. **Please note:** you need to put a tab character at the beginning of every command line! This is a common mistake that even experienced makefile writers can make.

Usually a command is defined by a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies commands for the target need not have prerequisites. For example, the rule containing the delete command associated with the target '`clean`' does not have prerequisites.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

17.5.2 A simple makefile

Here is a straightforward makefile that describes the way an executable file called '`edit`' depends on eight object files which, in turn, depend on eight C source files and three header files.

In this example, all the C files include '`defs.h`', but only files that define editing commands include '`command.h`', and only low-level files that change the editor buffer include '`buffer.h`'.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

We split each long line into two lines using a backslash; this is like using one long line, but easier to read.

To use this makefile to create the executable file called '`edit`', type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

In the example makefile, the targets include the executable file '`edit`', and the object files '`main.o`' and '`kbd.o`'. The prerequisites are files such as '`main.c`' and '`defs.h`'. In fact, each '`.o`' file is both a target and a prerequisite. Commands include `cc -c main.c` and `cc -c kbd.c`.

When a target is a file, it needs to be recompiled or relinked if any of its prerequisites change. In addition, any prerequisites that are themselves automatically generated should be updated first. In this example, '`edit`' depends on each of the eight object files; the object file '`main.o`' depends on the source file '`main.c`' and on the header file '`defs.h`'.

A shell command follows each line that contains a target and prerequisites. These shell commands tell `make` how to update the target file. A tab character must come at the beginning of every command line to distinguish command lines from other lines in the makefile. (Bear in mind that `make` does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly.)

The target `clean` is not a file, but merely the name of an action. Since this action is not carried out as part of the other targets, `clean` is not a prerequisite of any other rule. Consequently, `make` never does anything with it unless you explicitly type `make clean`. Not only is this rule *not* a prerequisite, it does not have any prerequisites itself, so the only purpose of the rule is to run the specified commands. Targets like `clean` that do not refer to files but are just actions are called *phony targets*.

17.5.3 make in action

By default, `make` starts with the first target whose name does not start with '`.`'. This is called the *default goal*. (*Goals* are the targets that `make` tries to update.)

In the simple example of the previous section, the default goal is to update the executable program '`edit`'; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking '`edit`'; but before `make` can fully process this rule, it must process the rules for the files that '`edit`' depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything that the goal depends on, and so forth), then that rule is not processed, unless you tell `make` to do so (with a command such as `make clean`.

Before recompiling an object file, `make` considers updating its prerequisites (the source file and header files). This makefile does not specify anything to be done for them—the '`.c`' and '`.h`' files are not the targets of any rules—so `make` does nothing for these files. But `make` can update automatically generated C programs, such as those made by Bison or Yacc, by defining the '`.c`' and '`.h`' files as targets and specifying how to create them with Bison, Yacc, or whatever other program generated them.

After recompiling the appropriate object files, `make` decides whether to link '`edit`'. This must be done if the file '`edit`' does not exist, or if any of the object files are newer than it is. If an object file was just recompiled, it is now newer than '`edit`', so '`edit`' is relinked.

Thus, if we change the file '`insert.c`' and run `make`, then `make` will recompile that file, update '`insert.o`', and then link '`edit`'. If we change the file '`command.h`' and run `make`, `make` will recompile the object files '`kbd.o`', '`command.o`' and '`files.o`', and then link the file '`edit`'.

17.5.4 Variables simplify makefiles

In our example, we had to list all the object files twice in the rule for '`edit`' (repeated here):

```
edit : main.o kbd.o command.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. *Variables* in `make` enable a text string to be defined once and substituted in multiple places later. They are similar to C macros. (See Section 12.2 [Macros], page 72.)

It is standard practice for every makefile to have a variable named `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, or `OBJ` that is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Then, in every place we want to put a list of the object file names, we can substitute the variable's value by writing `$(objects)`

Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

17.5.5 Letting make deduce commands

It is not necessary to spell out the commands for compiling the individual C source files, because `make` can figure them out: it has an *implicit rule* for updating a '`.o`' file from a correspondingly named '`.c`' file using a `gcc -c` command. For example, it will use the command `gcc -c main.c -o main.o` to compile '`main.c`' into '`main.o`'. We can therefore omit the commands from the rules for the object files.

When a '`.c`' file is used automatically in this way, it is also automatically added to the list of prerequisites. We can therefore omit the '`.c`' files from the prerequisites, provided we omit the commands.

Here is the entire example, with both of these changes, and the variable `objects` as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

This is how we would write the makefile in actual practice. (See Section 17.5.7 [Rules for cleaning the directory], page 172, for the complications associated with `clean`.)

Because implicit rules are so convenient, they are important. You will see them used frequently.

17.5.6 Combining rules by prerequisite

When the objects of a makefile are created by implicit rules alone, an alternative style of makefile is possible. In this style of makefile, you group entries by their prerequisites instead of by their targets. Here is an example of this alternative style:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here '`defs.h`' is given as a prerequisite of all the object files, and '`command.h`' and '`buffer.h`' are prerequisites of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

17.5.7 Rules for cleaning the directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly do a few other things besides compiling a program: for example, they can often delete all the object files and executables so that the directory is *clean*.

Here is how we could write a `make` rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. For example:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

This prevents `make` from getting confused by an actual file called '`clean`' and causes it to continue in spite of errors from `rm`.

A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for '`edit`', which recompiles the editor, to remain the default goal.

Since `clean` is not a prerequisite of '`edit`', this rule will not run at all if we give the command `make` with no arguments. In order to run the rule, we have to type `make clean`.

17.6 Building a library

We explored what libraries are and how to use them in a previous chapter. (See Chapter 13 [Libraries], page 79, if you need to refresh your memory.) You may have wondered how libraries are written in the first place. Is the whole process too complicated for a mortal C programmer to attempt? Not at all.

Suppose you have a function (or set of functions) that you would like to use widely across the various C programs you write. You might even like to make it available to other users in a convenient way. To create a code library that will enable you to achieve this, follow the sequence below. We will use a code example, but you can create your own library by taking similar steps.

1.

   Here's an example of the kind of function you might like to use in multiple programs. It accepts one string containing some text to print, and then prints it on the default printer.

   For the sake of example, the file below is named '`lpr_print.c`'.

   ```
   #include <stdio.h>

   void lpr_print (char *the_text)
   {
     FILE *printer;

     printer = popen ("lpr", "w");
     fprintf (printer, the_text);
     pclose (printer);
   }
   ```

   (See Section 16.4 [Programming with pipes], page 143, for the rationale behind this function.)

2.

   Now we will create a library.

   - To create a static library called '`liblprprint.a`' containing this function, just type the following two command lines in your GNU shell:

     ```
     gcc -c lpr_print.c
     ar rs liblprprint.a lpr_print.o
     ```

     The '`-c`' option to `gcc` produces only a '`.o`' object code file, without linking it, while the `ar` command (with its '`rs`' options) permits the creation of an *archive* file, which can contain a bundle of other files that can be re-extracted later (for example, when executing library code). In this case, we are only archiving one object code file, but in some cases, you might want to archive multiple ones. (See the `man` page for `ar` for more information.)

   - To create a shared library called '`liblprprint.so`' instead, enter the following sequence of commands:[1]

     ```
     gcc -c -fpic lpr_print.c
     gcc -shared -o liblprprint.so lpr_print.o
     ```

     (For the record, '`pic`' stands for "position-independent code", an object-code format required for shared libraries. You might need to use the option '`-fPIC`' instead of '`-fpic`' if your library is very large.)

3. Now create a header file that will allow users access to the functions in your library. You should provide one function prototype for each function in your library. Here is a header file for the library we have created, called '`liblprprint.h`'.

---

[1] To create library files containing multiple object files, simply include the object files on the same command line. For example, to create a static library with multiple object files, type a command such as `ar rs liblprprint.a lpr_print.o lpr_print2.o lpr_print3.o`. Similarly, to create a shared library, type `gcc -shared -o liblprprint.so lpr_print.o lpr_print2.o lpr_print3.o`.

```
/*
   liblprprint.h:
   routines in liblprprint.a
   and liblprprint.so
*/

extern void lpr_print (char *the_text);
```

4. Now you should put your libraries and include file somewhere your code can access them. For the sake of this example, create the directories 'include' and 'lib' in your home directory. Once you have done so, move the '.a' and '.so' files you have created to 'lib', and the '.h' file to 'include'.

5. If you have taken the last step, and you want to run a program linked to a shared version of your library, you should type a line like the following into your shell (the following command line assumes you are using the Bash shell and that your home directory is named '/home/fred'):

```
export LD_LIBRARY_PATH=/home/fred/lib:$LD_LIBRARY_PATH
```

This command line sets an environment variable that makes the linker search the '/home/fred/lib' directory before it searches anywhere else. You can include it in your '.bashrc' or '.bash_profile' file. If you don't execute this command before you attempt to run a program using your shared library, you will probably receive an error.

6. Now you can write programs that use your library. Consider the following short program, called 'printer.c':

```
#include <liblprprint.h>

/* To shorten example, not using argp */
int main ()
{
  lpr_print ("Hello, Multiverse!\nHowarya?\n");
  return 0;
}
```

To compile this program using your static library, type something like the following command line:

```
gcc --static -I../include -L../lib -o printer printer.c -llprprint
```

The '--static' option forces your static library to be linked; the default is your shared version. The '-llprprint' option makes GCC link in the 'liblprprint' library, just as you would need to type '-lm' to link in the 'libm' math library.

The '-I../include' and '-L../lib' options specify that the compiler should look in the '../include' directory for include files and in the '../lib' directory for library files. This assumes that you have created the 'include' and 'lib' directories in your home directory as outlined above, and that you are compiling your code in a subdirectory of your home directory. If you are working two directories down, you would specify '-I../../include', and so on.

The above command line assumes you are using only one '.c' source code file; if you are using more than one, simply include them on the command line as well. (See Section 17.4 [Compiling multiple files], page 166.)

Note: Using the '--static' option will force the compiler to link all libraries you are using statically. If you want to use the static version of your library, but some shared versions of other libraries, you can omit the '--static' option from the command line and specify the static version of your library explicitly, as follows:

```
gcc -I../include -L../lib -o printer printer.c ../lib/liblprprint.a
```

- To compile this program using your shared library, type something like the following command line.

```
gcc -I../include -L../lib -o printer printer.c -llprprint
```

7. The executable produced is called 'printer'. Try it!

17.7 Questions

1. What is the name of the preferred method for handling command-line options?
2. What does the '-c' option of the gcc command do?
3. What information does the argc variable contain?
4. What information does the argv variable contain?
5. What information does the envp variable contain?

18 Advanced operators

Concise expressions

In this chapter, we will examine some advanced mathematical and logical operators in C.

18.1 Hidden operators and values

Many operators in C are more versatile than they appear to be at first glance. Take, for example, the following operators:

- `=`
- `++`
- `--`
- `+=`
- `-=`

These operators can be used in some surprising ways to make C source code elegant and compact. (See Chapter 7 [Expressions and operators], page 31, if you need a refresher in what they do.) All of them can form expressions that have their own values. Such an expression can be taken as a whole (a "black box") and treated as a single value, which can then be assigned and compared to other expressions, in effect, "hidden" within another expression.

The value of an expression is the result of the operation carried out in the expression. Increment and decrement statements have a value that is one greater than or one less than the value of the variable they act upon, respectively.

Consider the following two statements:

```
c = 5;
c++;
```

The expression `c++` in the above context has the value 6.

Now consider these statements:

```
c = 5;
c--;
```

The expression `c--` in the above context has the value 4.

18.1.1 Hidden assignments

Assignment expressions have values too — their values are the value of the assignment. For example, the value of the expression `c = 5` is 5.

The fact that assignment statements have values can be used to make C code more elegant. An assignment expression can itself be assigned to a variable. For example, the expression `c = 0` can be assigned to the variable `b`:

```
b = (c = 0);
```

or simply:

```
b = c = 0;
```

These equivalent statements set b and c to the value 0, provided b and c are of the same type. They are equivalent to the more usual:

```
b = 0;
c = 0;
```

Note: Don't confuse this technique with a logical test for equality. In the above example, both b and c are set to 0. Consider the following, superficially similar, test for equality, however:

```
b = (c == 0);
```

In this case, b will only be assigned a zero value (FALSE) if c does not equal 0. If c does equal 0, then b will be assigned a non-zero value for TRUE, probably 1. (See Section 7.8 [Comparisons and logic], page 36, for more information.)

Any number of these assignments can be strung together:

```
a = (b = (c = (d = (e = 5))));
```

or simply:

```
a = b = c = d = e = 5;
```

This elegant syntax compresses five lines of code into a single line.

There are other uses for treating assignment expressions as values. Thanks to C's flexible syntax, they can be used anywhere a value can be used. Consider how an assignment expression might be used as a parameter to a function. The following statement gets a character from standard input and passes it to a function called process_character.

```
process_character (input_char = getchar());
```

This is a perfectly valid statement in C, because the hidden assignment statements passes the value it assigns on to process_character. The assignment is carried out first and then the process_character function is called, so this is merely a more compact way of writing the following statements.

```
input_char = getchar();
process_character (input_char);
```

All the same remarks apply about the specialized assignment operators +=, *=, /=, and so on.

The following example makes use of a hidden assignment in a while loop to print out all values from 0.2 to 20.0 in steps of 0.2.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main ()
{
  double my_dbl = 0;

  while ((my_dbl += 0.2) < 20.0)
    printf ("%lf ", my_dbl);

  printf ("\n");

  return 0;
}
```

18.1.2 Postfix and prefix ++ and --

Increment (`++`) and decrement (`--`) expressions also have values, and like assignment expressions, can be hidden away in inconspicuous places. These two operators are slightly more complicated than assignments because they exist in two forms, postfix (for example, `my_var++`) and prefix (for example, `++my_var`).

Postfix and prefix forms have subtly different meanings. Take the following example:

```
int my_int = 3;
printf ("%d\n", my_int++);
```

The increment operator is hidden in the parameter list of the `printf` call. The variable `my_int` has a value before the `++` operator acts on it (3) and afterwards (4).

Which value is passed to `printf`? Is `my_int` incremented before or after the `printf` call? This is where the two forms of the operator (postfix and prefix) come into play.

If the increment or decrement operator is used as a prefix, the operation is performed *before* the function call. If the operator is used as a postfix, the operation is performed *after* the function call.

In the example above, then, the value passed to `printf` is 3, and when the `printf` function returns, the value of `my_int` is incremented to 4. The alternative is to write

```
int my_int = 3;
printf ("%d\n", ++my_int);
```

in which case the value 4 is passed to `printf`.

The same remarks apply to the decrement operator as to the increment operator.

18.1.3 Arrays and hidden operators

Hidden operators can simplify dealing with arrays and strings quite a bit. Hiding operators inside array subscripts or hidding assignments inside loops can often streamline tasks such as array initialization. Consider the following example of a one-dimensional array of integers.

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

```
#define ARRAY_SIZE 20

/* To shorten example, not using argp */
int main ()
{
  int idx, array[ARRAY_SIZE];

  for (idx = 0; idx < ARRAY_SIZE; array[idx++] = 0)
    ;

  return 0;
}
```

This is a convenient way to initialize an array to zero. Notice that the body of the loop is completely empty!

Strings can benefit from hidden operators as well. If the standard library function `strlen`, which finds the length of a string, were not available, it would be easy to write it with hidden operators:

```
#include <stdio.h>

int my_strlen (char *my_string)
{
  char *ptr;
  int count = 0;

  for (ptr = my_string; *(ptr++) != '\0'; count++)
    ;

  return (count);
}
/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{
  char string_ex[] = "Fabulous!";

  printf ("String = '%s'\n", string_ex);
  printf ("Length = %d\n", my_strlen (string_ex));

  return 0;
}
```

The `my_strlen` function increments `count` while the end of string marker '`\0`' is not found. Again, notice that the body of the loop in this function is completely empty.

18.1.4 A warning about style

Overuse of "hidden" operators can produce code that is difficult to understand. See Section 22.6 [Hidden operators and style], page 221, for some cautions about when not to use them.

18.2 The comma operator

The comma operator (`,`) works almost like the semicolon '`;`' that separates one C statement from another. You can separate almost any kind of C statment from another with a comma operator. The comma-separated expressions are evaluated from left to right and the value of the whole comma-separated sequence is the value of the rightmost expression in the sequence. Consider the following code example.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{
  int a, b, c, d;

  a = (b = 2, c = 3, d = 4);
  printf ("a=%d\nb=%d\nc=%d\nd=%d\n",
  a, b, c, d);
  return 0;
}
```

The value of `(b = 2, c = 3, d = 4)` is 4 because the value of its rightmost sub-expression, `d = 4`, is 4. The value of `a` is thus also 4. When run, this example prints out the following text:

```
a=4
b=2
c=3
d=4
```

The comma operator is very useful in `for` loops. (See Section 11.4 [The flexibility of for], page 65, for an example.)

18.3 Machine-level operators

Bits and Bytes. Flags. Shifting.

Bits (or binary digits), the values 0 and 1, are the lowest-level software objects in a computer; there is nothing more primitive. C gives programmers full access to bits and bit sequences, and even provides high-level operators for manipulating them.

All computer data whatsoever is composed of *bit strings*. The word "string" is being used here in its more general sense of sequence; do not confuse the usage with "text string". Although all text strings are bit strings, not all bit strings are text strings.

The only difference between a text string and a floating-point value is the way we interpret the pattern of bits in the computer's memory. For the most part, we can simply ignore the low level of the computer in which computer data appears as bit strings. Systems programmers, on the other hand, such as those who wrote GNU/Linux, must frequently manipulate bits directly in order to handle flags.

A *flag* is a bit in a bit string that can be either *set* (1) or *cleared* (0). We have run across a few flags already, such as the various flags passed to the GNU C Library functions `open`; the flags `O_RDONLY` and `O_WRONLY` are actually macros that specify binary values, which can be manipulated and examined with binary OR and similar functions. Flags are normally declared as integers in C.

Programmers who perform bit operations on a regular basis often use either octal (base-8) or hexadecimal (base-16) numbers, because every octal digit specifies exactly three bits, and every hexadecimal digit specifies four.

18.3.1 Bitwise operators

C provides the following operators for handling bit patterns:

`<<` Bit-shift left by a specified number of bit positions

`>>` Bit-shift right by a specified number of bit positions

`|` Bitwise inclusive OR

`^` Bitwise exclusive OR

`&` Bitwise AND

`~` Bitwise NOT

`<<=` Bit-shift left assignment (*var* = *var* `<<` *value*)

`>>=` Bit-shift right assignment (*var* = *var* `>>` *value*)

`|=` Exclusive OR assignment (*var* = *var* `|` *value*)

`^=` Inclusive OR assignment (*var* = *var* `^` *value*)

`&=` AND assignment (*var* = *var* & *value*)

The meaning and syntax of these operators is given below.

Don't confuse bitwise operators (such as bitwise AND, `&`) with logical operators (such as logical AND, `&&`). Bitwise operators operate on each bit in the operand individually.

18.3.2 Shift operations

Imagine a bit string as represented by the following group of boxes. Every box represents a bit, a binary digit; the ones and zeros inside represent their values. The values written across the top are the place-values of each bit. (Just as a decimal (base-10) number has a ones place, a tens place, a hundreds place, a thousands place, and so on, a binary (base-2) number has the places 1, 2, 4, 8, 16, 32, etc.) The number after the equals sign shows the value of the bit string in decimal notation.

```
 128  64  32  16  8   4   2   1
 -------------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |   =  1
 -------------------------------
```

Bit-shift operators move whole bit strings left or right. The syntax of the bit-shift left operation is *value* `<<` *positions*; that of bit-shift right is *value* `>>` *positions*; So for example, using the bit string (1) above, the value of `1 << 1` is 2, because the bit string would have been moved one position to the left:

```
128  64  32  16  8   4   2   1
-------------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |   =  2
-------------------------------
```

Notice how the space to the right of the shifted bit string is simply filled with a 0.

Similarly, the value of `1 << 4` is 16, because the original bit string is shifted left four places:

```
128  64  32  16  8   4   2   1
-------------------------------
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |   =  16
-------------------------------
```

Notice, again, that the spaces to the right of the original bit string are filled out with zeros.

Now for a slightly more difficult one. The value of `6 << 2` is 24. Here is the bit string representing 6:

```
128  64  32  16  8   4   2   1
-------------------------------
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |   =  6
-------------------------------
```

Shift 6 left 2 places:

```
128  64  32  16  8   4   2   1
-------------------------------
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |   =  24
-------------------------------
```

Notice that every shift left multiplies by 2. (Since `6 << 2` means to shift 6 left twice, the result is 24.)

As you might expect, every shift right performs (integer) division by two on the number. If a bit is shifted beyond the ones position (the rightmost "box"), however, then it "drops off" and is lost. So the following equalities hold:

```
1 >> 1 == 0
2 >> 1 == 1
2 >> 2 == 0
n >> n == 0
```

One common use of bit-shifting is to scan through the bits of a bit-string one by one in a loop. This is done with bit masks, as described in the next section.

18.3.3 Truth tables and bit masks

The binary operators AND (`&`), OR (inclusive OR, `|`) and XOR (exclusive OR, also called EOR, `^`) perform comparisons, or *masking* operations, between two bit strings. They are also binary operators in the sense that they take two operands. There is another operator called NOT (`~`) that is a unary operator; it takes only one operand.

These bitwise operations are best summarized by *truth tables*. Each truth table for a binary operator (that is, one with two operands), indicates what the result of the operation is for every possible combination of two bits.

18.3.3.1 Bitwise NOT

The unary operator NOT (`~`) simply generates the *one's complement* of the bit string; that is, it returns the same bit string, with all ones replaced with zeros and all zeros replaced with ones. As a truth table this would be summarized as follows:

~value	*result*
0	1
1	0

18.3.3.2 Bitwise AND

Bitwise AND operates on two values, for example `0 & 1`. Both the first value *and* the second value must be 1 in order for the result to be 1. As a truth table this would be summarized as follows:

value1	*value2*	*result*
0	0	0
0	1	0
1	0	0
1	1	1

18.3.3.3 Bitwise inclusive OR

Bitwise OR operates on two values, for example `0 | 1`. The result is 1 if the first value *or* the second value is 1, *or* both are 1. As a truth table this would be summarized as follows:

value1	*value2*	*result*
0	0	0
0	1	1
1	0	1
1	1	1

18.3.3.4 Bitwise exclusive OR (XOR/EOR)

Bitwise XOR operates on two values, for example `0 ^ 1`. The result is 1 if the first value *or* the second value is 1, but *not* if *both* are 1 (hence the name "exclusive OR"). As a truth table this would be summarized as follows:

value1	*value2*	*result*
0	0	0
0	1	1
1	0	1
1	1	0

18.3.3.5 Masks

Bit strings and bitwise operators are often used to make *masks*. A mask is a bit string that "fits over" another bit string and produces a desired result, such as singling out particular bits from the second bit string, when the two bit strings are operated upon. This is particularly useful for handling flags; programmers often wish to know whether one particular flag is set in a bit string, but may not care about the others. For example, you might create a mask that only allows the flag of interest to have a non-zero value, then AND that mask with the bit string containing the flag.

Consider the following mask, and two bit strings from which we want to extract the final bit:

```
mask   = 00000001
value1 = 10011011
value2 = 10011100

mask & value1 == 00000001
mask & value2 == 00000000
```

The zeros in the mask *mask off* the first seven bits and only let the last bit show through. (In the case of the first value, the last bit is 1; in the case of the second value, the last bit is 0.)

Alternatively, masks can be built up by operating on several flags, usually with inclusive OR:

```
flag1 = 00000001
flag2 = 00000010
flag3 = 00000100

mask = flag1 | flag2 | flag3

mask == 00000111
```

See Section 16.5.2 [Opening files at a low level], page 147, for a code example that actually uses bitwise OR to join together several flags.

It should be emphasized that the flag and mask examples are written in *pseudo-code*, that is, a means of expressing information that resembles source code, but cannot be compiled. It is not possible to use binary numbers directly in C.

The following code example shows how bit masks and bit-shifts can be combined. It accepts a decimal number from the user between 0 and 128, and prints out a binary number in response.

```
#include <stdio.h>
#define NUM_OF_BITS 8

/* To shorten example, not using argp */
int main ()
{
  char *my_string;
  int input_int, args_assigned;
  int nbytes = 100;
  short my_short, bit;
  int idx;

  /* This hex number is the same as binary 10000000 */
  short MASK = 0x80;

  args_assigned = 0;
  input_int = -1;

  while ((args_assigned != 1) ||
 (input_int < 0) || (input_int > 128))
    {
      puts ("Please enter an integer from 0 to 128.");
      my_string = (char *) malloc (nbytes + 1);
      getline (&my_string, &nbytes, stdin);
      args_assigned = sscanf (my_string, "%d", &input_int);
      if ((args_assigned != 1) ||
  (input_int < 0) || (input_int > 128))
puts ("\nInput invalid!");
    }

  my_short = (short) input_int;

  printf ("Binary value = ");

  /*
     Convert decimal numbers into binary
     Keep shifting my_short by one to the left
     and test the highest bit.  This does
     NOT preserve the value of my_short!
  */

  for (idx = 0; idx < NUM_OF_BITS; idx++)
    {
      bit = my_short & MASK;
      printf ("%d", bit/MASK);
      my_short <<= 1;
    }

  printf ("\n");
  return 0;
}
```

18.4 Questions 18

1. Hidden operators can be used in return statements, for example,

   ```
   return (++x);
   ```

   Would there be any point in writing the following?

   ```
   return (x++);
   ```

2. What distinguishes a bit string from an ordinary variable? Can any variable be a bit string?
3. What is the difference between an inclusive OR operation and an exclusive OR operation?
4. Find out what the decimal values of the following operations are.
   1. `7 & 2`
   2. `1 & 1`
   3. `15 & 3`
   4. `15 & 7`
   5. `15 & 7 & 3`

   Try to explain the results. (Hint: sketch out the numbers as bit strings.)
5. Find out what the decimal values of the following operations are.
   1. `1 | 2`
   2. `1 | 2 | 3`
6. Find out the decimal values of the following operations.
   1. `1 & (~1)`
   2. `23 & (~23)`
   3. `2012 & (~2012)`

   (Hint: write a short program to work them out.)

19 More data types

There are still a few data types in C that we have not discussed. Actually, since C allows you to define new data types at will, no one can ever cover all possibilities. We will only discuss the most important examples.

`enum` Type specifier for variables that can have a set of different values.

`void` Type specifier for "empty" data.

`volatile` Type qualifier for data that changes independently of the program.

`const` Type qualifier for data that cannot change.

In addition, there are two data types called `struct` and `union` that are so important, they have received their own chapter. (See Chapter 20 [Data structures], page 197, for more information on `struct` and `union`.)

19.1 enum

The `enum` type specifier is short for "enumerated data". The user can define a fixed set of words that a variable of type `enum` can take as its value. The words are assigned integer values by the compiler so that code can compare `enum` variables. Consider the following code example:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main ()
{
  enum compass_direction
  {
    north,
    east,
    south,
    west
  };

  enum compass_direction my_direction;
  my_direction = west;

  return 0;
}
```

This example defines an enumerated variable type called `compass_direction`, which can be assigned one of four enumerated values: `north`, `east`, `south`, or `west`. It then declares a variable called `my_direction` of the enumerated `compass_direction` type, and assigns `my_direction` the value `west`.

Why go to all this trouble? Because enumerated data types allow the programmer to forget about any numbers that the computer might need in order to process a list of words, and simply concentrate on using the words themselves. It's a higher-level way of doing things; in fact, at a lower level, the computer assigns each possible value in an enumerated data type an integer cconstant — one that you do not need to worry about.

Enumerated variables have a natural partner in the `switch` statement, as in the following code example.

```
#include <stdio.h>

enum compass_direction
{
  north,
  east,
  south,
  west
};


enum compass_direction get_direction()
{
  return south;
}

/* To shorten example, not using argp */
int main ()
{
  enum compass_direction my_direction;
  puts ("Which way are you going?");
  my_direction = get_direction();

  switch (my_direction)
  {
    case north:
      puts("North? Say hello to the polar bears!");
      break;

    case south:
      puts("South? Say hello to Tux the penguin!");
      break;

    case east:
      puts("If you go far enough east, you'll be west!");
      break;

    case west:
      puts("If you go far enough west, you'll be east!");
      break;
  }

  return 0;
}
```

In this example, the `compass_direction` type has been made global, so that the `get_direction` function can return that type. The `main` function prompts the user, '`Which way are you going?`', then calls the "dummy" function `get_direction`. In a "real" program, such a function would accept input from the user and return an enumerated value to `main`, but in this case it merely returns the value `south`. The output from this code example is therefore as follows:

```
Which way are you going?
South? Say hello to Tux the penguin!
```

As mentioned above, enumerated values are converted into integer values internally by the compiler. It is practically never necessary to know what integer values the compiler assigns to the enumerated words in the list, but it may be useful to know the order of the enumerated items with respect to one another. The following code example demonstrates this.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main ()
{
  enum planets
  {
    Mercury,
    Venus,
    Earth,
    Mars,
    Jupiter,
    Saturn,
    Uranus,
    Neptune,
    Pluto
  };

  enum planets planet1, planet2;

  planet1 = Mars;
  planet2 = Earth;

  if (planet1 > planet2)
    puts ("Mars is farther from the Sun than Earth is.");
  else
    puts ("Earth is farther from the Sun than Mars is.");

  return 0;
}
```

The output from this example reads as follows:

```
Mars is farther from the Sun than Earth is.
```

19.2 void

The `void` data type was introduced to make C syntactically consistent. The main reason for `void` is to declare functions that have no return value. The word “void” is therefore used in the sense of “empty” rather than that of “invalid”.

C functions are considered by the compiler to return type `int` unless otherwise specified. Although the data returned by a function can legally be ignored by the function calling it, the `void` data type was introduced by the ANSI standard so that C compilers can issue warnings when an integer value is not returned by a function that is supposed to return one. If you want to write a function that does not return a value, simply declare it `void`. A function declared `void` has no return value and simply returns with the command `return;`.

Variables can be declared `void` as well as functions:

```
void my_variable;
void *my_pointer;
```

A variable that is itself declared `void` (such as `my_variable` above) is useless; it cannot be assigned a value, cannot be cast to another type, in fact, cannot be used in any way.

Void pointers (type `void *`) are a different case, however. A void pointer is a *generic pointer*; any pointer can be cast to a void pointer and back without any loss of information. Any type of pointer can be assigned to (or compared with) a void pointer, without casting the pointer explicitly.

Finally, a function call can be cast to `void` in order to explicitly discard a return value. For example, `printf` returns a value, but it is seldom used. Nevertheless, the two lines of code that follow are equivalent:

```
printf ("Hullo!\n");
(void) printf ("Hullo!\n");
```

There is no good reason to prefer the second line to the first, however, so using the more concise form is preferred.

19.3 `volatile`

The `volatile` type qualifier was introduced by the ANSI Standard to permit the use of *memory-mapped variables*, that is, variables whose value changes autonomously based on input from hardware. One might declare a volatile variable `volatile float temperature;` whose value fluctuated according to readings from a digital thermometer connected to the computer.

There is another use for the `volatile` qualifier that has to do with multiprocessing operating systems. Independent processes that share common memory might each change the value of a variable independently. The `volatile` keyword serves as a warning to the compiler that it should not *optimize* the code containing the variable (that is, compile it so that it will run in the most efficient way possible) by storing the value of the variable and referring to it repeatedly, but should reread the value of the variable every time. (Volatile variables are also flagged by the compiler as not to be stored in read-only memory.)

19.4 Constants

Constants in C usually refer to two things: either a type of variable whose value cannot change declared with the `const` qualifier (in this case, "variable" is something of a misnomer), or a string or numeric value incorporated directly into C code, such as '`1000`'. We will examine both kinds of constant in the next two sections.

19.4.1 `const`

Sometime a variable must be assigned a value once and once only; for example, it might be in read-only memory. The reserved word `const` is, like `static` and `volatile`, a data type qualifier that can be applied to many different data types. It declares a variable to be a constant, whose value cannot be reassigned. A `const` must be assigned a value when it is declared.

```
const double avogadro = 6.02e23;
const int moon_landing = 1969;
```

You can also declare constant arrays:

```
const int my_array[] =
{0, 1, 2, 3, 4, 5, 6, 7, 8};
```

Any attempt to assign a new value to a `const` variable will result in a compile-time error such as the following:

```
const.c: In function ‘main’:
const.c:11: warning: assignment of read-only variable ‘avogadro’
```

19.4.2 Constant expressions

You can declare constant expressions explicitly as a particular type of value, such as a long integer, a float, a character, or a hexadecimal value, with certain typographical conventions. For example, it is possible to declare a value explicitly as a long by placing the letter ‘`L`’ after the numeric constant. For example:

```
#define MY_LONG1 23L;
#define MY_LONG2 236526598L;
```

Similarly, you can declare a value to be a float by appending the letter ‘`F`’ to it. Of course, numeric constants containing a decimal point are automatically considered floats. The following constants are both floating-point numbers:

```
#define MY_FLOAT1 23F;
#define MY_FLOAT2 23.5001;
```

You can declare a hexadecimal (base-16) number by prefixing it with ‘`0x`’; you can declare an octal (base-8) number by prefixing it with ‘`0`’. For example:

```
int my_hex_integer = 0xFF;   /* hex FF */
int my_octal_integer = 077;  /* octal 77 */
```

You can use this sort of notation with strings and character constants too. ASCII character values range from 0 to 255. You can print any character in this range by prefixing a hexadecimal value with ‘`\x`’ or an octal value with ‘`\`’. Consider the following code example, which demonstrates how to print the letter ‘`A`’, using either a hexadecimal character code (‘`\x41`’) or an octal one (‘`\101`’).

```
#include <stdio.h>

/* To shorten example, not using argp */
int main ()
{
  printf ("\\x41 hex   = \x41\n");
  printf ("\\101 octal = \101\n");

  return 0;
}
```

The preceding code prints the following text:

```
\x41 hex   = A
\101 octal = A
```

Of course, you can assign a variable declared with the `const` qualifier (the first kind of "constant" we examined) a constant expression declared with one of the typographical expressions above. For example:

```
const int my_hex_integer = 0xFF;   /* hex FF */
const int my_octal_integer = 077;  /* octal 77 */
```

19.5 struct and union

Structures and unions are data types that are important enough to merit a chapter of their own. See Chapter 20 [Data structures], page 197, for more information on structures and unions.

19.6 typedef

You can define your own data types in C with the `typedef` command, which may be written inside functions or in global scope. This statement is used as follows:

`typedef` *existing_type new_type*`;`

You can then use the new type to declare variables, as in the following code example, which declares a new type called `my_type` and declares three variables to be of that type.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{
  typedef int my_type;
  my_type var1, var2, var3;

  var1 = 10;
  var2 = 20;
  var3 = 30;

  return 0;
}
```

The new type called `my_type` behaves just like an integer. Why, then, would we use it instead of `integer`?

Actually, you will seldom wish to rename an existing data type. The most important use for `typedef` is in renaming structures and unions, whose names can become long and tedious to declare otherwise. We'll investigate structures and unions in the next chapter. (See Chapter 20 [Data structures], page 197.)

19.7 Questions 19

1. Enumerated names are given integer values by the compiler so that it can do multiplication and division with them. True or false?
2. Does `void` do anything which C cannot already do without this type?
3. What type qualifier might a variable accessed directly by a timer be given?
4. Write a statement which declares a new type `"real"` to be like the usual type `"double"`.
5. Variables declared with the qualifier `const` can be of any type. True or false?

20 Data structures

Grouping data. Tidying up programs.

It would be hard for a program to manipulate data if it were scattered around with no particular structure. C therefore has several facilities to group data together in convenient packages, or *data structures*. One type of data structure in C is the `struct` (or *structure*) data type, which is a group of variables clustered together with a common name. A related data type is the `union`, which can contain any type of variable, but only one at a time. Finally, structures and unions can be linked together into complex data structures such as lists and trees. This chapter explores all of these kinds of data structure.

It is important to distinguish the terms *structure* and *data structure*. "Data structure" is a generic term that refers to any pattern of data in a computer program. An array is a data structure, as is a string. A structure is a particular data type in C, the `struct`; all `struct` variables (structures) are data structures, but not all data structures are structures.

20.1 `struct`

A *structure* is a group of one or more variables under a single name. Unlike arrays, structures can contain a combination of different types of data; they can even contain arrays. A structure can be arbitrarily complex.

Every type of structure that is defined is given a name, and the variables it contains (called *members*) are also given names. Finally, every variable declared to be of a particular structure type has its own name as well, just as any other variable does.

20.1.1 Structure declarations

The following statement is a type declaration, so it belongs with other declarations, either at the start of a program or the start of a code block.

```
struct personal_data
{
  char name[100];
  char address[200];
  int year_of_birth;
  int month_of_birth;
  int day_of_birth;
};
```

The statement says: define a type of variable that holds a string of 100 characters called `name`, a string of 200 characters called `address`, and three integers called `year_of_birth`, `month_of_birth`, and `day_of_birth`. Any variable declared to be of type `struct personal_data` will contain these components, which are called *members*. Different structures, even different *types* of structure, can have members with the same name, but the values of members of different structures are independent of one another. You can also use the same name for a member as for an ordinary variable in your program, but the computer will recognize them as different entities, with different values. This is similar to the naming convention for humans, where two different men may share the name "John Smith", but are recognized as being different people.

Once you have declared a type of structure, you can declare variables to be of that type. For example:

```
struct personal_data person0001;
```

The statement above declares a variable called `person0001` to be of type `struct personal_data`. This is probably the most common method of declaring a structure variable, but there are two equivalent methods. For example, a structure variable can be declared immediately after the declaration of the structure type:

```
struct personal_data
{
  char name[100];
  char address[200];
  int year_of_birth;
  int month_of_birth;
  int day_of_birth;
} person0001;
```

20.1.1.1 Structure declarations using typedef

Alternatively, the `typedef` command can be used to cut down on typing out code in the long term. The type definition is made once at the start of the program and subsequent variable declarations are made by using the new name, without the word `struct`:

```
typedef struct
{
  char name[100];
  char address[200];
  int year_of_birth;
  int month_of_birth;
  int day_of_birth;
} personal_data;

personal_data person001;
personal_data person002;
personal_data person003;
```

Note that this use of the `typedef` command parallels the usage we have already seen:

`typedef` *existing_type new_type*

In the example above of using `typedef` to declare a new type of structure, the metasyntactic variable *new_type* corresponds to the identifier `personal_data`, and the metasyntactic variable *existing_type* corresponds to the following code:

```
struct
{
  char name[100];
  char address[200];
  int year_of_birth;
  int month_of_birth;
  int day_of_birth;
}
```

Structure type and variable declarations can be either local or global, depending on their placement in the code, just as any other declaration can be.

20.1.2 Using structures

Structures are extremely powerful data types. Not only can you pass a whole structure as a parameter to a function, or return one as a value from a function. You can even assign one structure to another.

You can get and set the values of the members of a structure with the '.' dot character. This is called the *member operator*. The general form of a member reference is:

structure_name.*member_name*

In the following example, the year 1852 is assigned to the `year_of_birth` member of the structure variable `person1`, of type `struct personal_data`. Similarly, month 5 is assigned to the `month_of_birth` member, and day 4 is assigned to the `day_of_birth` member.

```
struct personal_data person1;

person1.year_of_birth = 1852;
person1.month_of_birth = 5;
person1.day_of_birth = 4;
```

Besides the dot operator, C also provides a special `->` member operator for use in conjunction with pointers, because pointers and structures are used together so often. (See Section 20.1.5 [Pointers to structures], page 201.)

Structures are easy to use For example, you can assign one structure to another structure of the same type (unlike strings, for example, which must use the string library routine `strcpy`). Here is an example of assigning one structure to another:

```
struct personal_data person1, person2;

person2 = person1;
```

The members of the `person2` variable now contain all the data of the members of the `person1` variable.

Structures are passed as parameters in the usual way:

```
my_structure_fn (person2);
```

You would declare such a function thus:

```
void my_structure_fn (struct personal_data some_struct)
{
}
```

Note that in order to declare this function, the `struct personal_data` type must be declared globally.

Finally, a function that returns a structure variable would be declared thusly:

```
struct personal_data structure_returning_fn ()
{
  struct personal_data random_person;
  return random_person;
}
```

Of course, `random_person` is a good name for the variable returned by this bare-bones function, because without unless one writes code to initialize it, it can only be filled with garbage values.

20.1.3 Arrays of structures

Just as arrays of basic types such as integers and floats are allowed in C, so are arrays of structures. An array of structures is declared in the usual way:

```
struct personal_data my_struct_array[100];
```

The members of the structures in the array are then accessed by statements such as the following:

The value of a member of a structure in an array can be assigned to another variable, or the value of a variable can be assigned to a member. For example, the following code assigns the number 1965 to the `year_of_birth` member of the fourth element of `my_struct_array`:

```
my_struct_array[3].year_of_birth = 1965;
```

(Like all other arrays in C, `struct` arrays start their numbering at zero.)

The following code assigns the value of the `year_of_birth` member of the fourth element of `my_struct_array` to the variable `yob`:

```
yob = my_struct_array[3].year_of_birth;
```

Finally, the following example assigns the values of all the members of the second element of `my_struct_array`, namely `my_struct_array[1]`, to the third element, so `my_struct_array[2]` takes the overall value of `my_struct_array[1]`.

```
my_struct_array[2] = my_struct_array[1];
```

20.1.4 Nested structures

Structures can contain other structures as members; in other words, structures can *nest*. Consider the following two structure types:

```
struct first_structure_type
{
  int integer_member;
  float float_member;
};

struct second_structure_type
{
  double double_member;
  struct first_structure_type struct_member;
};
```

The first structure type is incorporated as a member of the second structure type. You can initialize a variable of the second type as follows:

```
struct second_structure_type demo;

demo.double_member = 12345.6789;
demo.struct_member.integer_member = 5;
demo.struct_member.float_member = 1023.17;
```

The member operator '.' is used to access members of structures that are themselves members of a larger structure. No parentheses are needed to force a special order of evaluation; a member operator expression is simply evaluated from left to right.

In principle, structures can be nested indefinitely. Statements such as the following are syntactically acceptable, but bad style. (See Chapter 22 [Style], page 219.)

```
my_structure.member1.member2.member3.member4 = 5;
```

What happens if a structure contains an instance of its own type, however? For example:

```
struct regression
{
  int int_member;
  struct regression self_member;
};
```

In order to compile a statement of this type, your computer would theoretically need an infinite amount of memory. In practice, however, you will simply receive an error message along the following lines:

```
struct5.c: In function 'main':
struct5.c:8: field 'self_member' has incomplete type
```

The compiler is telling you that `self_member` has been declared before its data type, `regression` has been fully declared — naturally, since you're declaring `self_member` in the middle of declaring its own data type!

20.1.5 Pointers to structures

Although a structure cannot contain an instance of its own type, it can can contain a pointer to another structure of its own type, or even to itself. This is because a pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure. Pointers to structures are quite invaluable, in fact, for building data structures such as linked lists and trees. (See Section 20.4 [Complex data structures], page 206.)

A pointer to a structure type variable is declared by a statement such as the following:

```
struct personal_data *my_struct_ptr;
```

The variable `my_struct_ptr` is a pointer to a variable of type `struct personal_data`. This pointer can be assigned to any other pointer of the same type, and can be used to access the members of its structure. According to the rules we have outlined so far, this would have to be done like so:

```
struct personal_data person1;

my_struct_ptr = &person1;
(*my_struct_ptr).day_of_birth = 23;
```

This code example says, in effect, "Let the member `day_of_birth` of the structure pointed to by `my_struct_ptr` take the value 23." Notice the use of parentheses to avoid confusion about the precedence of the '*' and '.' operators.

There is a better way to write the above code, however, using a new operator: '->'. This is an arrow made out of a minus sign and a greater than symbol, and it is used as follows:

```
my_struct_ptr->day_of_birth = 23;
```

The '->' enables you to access the members of a structure directly via its pointer. This statement means the same as the last line of the previous code example, but is considerably clearer. The '->' operator will come in very handy when manipulating complex data structures. (See Section 20.4 [Complex data structures], page 206.)

20.1.6 Initializing structures

In the chapter on arrays, we explored how to initialize an array with values at compile time. (See Section 14.5 [Initializing arrays], page 97.) It is also possible to initialize structures at compile time, as shown below. (This code example also shows how to dynamically allocate structures with `malloc` and initialize them with the `->` operator. See Section 20.2 [Memory allocation], page 203, for more information on this technique.)

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  struct personal_data
  {
    char name[100];
    char address[200];
    int year_of_birth;
    int month_of_birth;
    int day_of_birth;
  };
  struct personal_data person1 =
  {
    "Liddell, Alice",
    "Wonderland",
    1852,
    5,
    4
  };
  struct personal_data person2 =
  {
    "Hale-Evans, Ron",
    "Seattle, Washington",
    1965,
    6,
    27
  };
```

```
  struct personal_data* person_ptr1;
  struct personal_data* person_ptr2;

  person_ptr1 = (struct personal_data*)
    malloc (sizeof (struct personal_data));

  strcpy (person_ptr1->name, "Adams, Douglas");
  strcpy (person_ptr1->address, "The Galaxy");
  person_ptr1->year_of_birth = 1952;
  person_ptr1->month_of_birth = 3;
  /* Don't know his exact birthday */
  person_ptr2 = (struct personal_data*)
    malloc (sizeof (struct personal_data));

  strcpy (person_ptr2->name, "Egan, Greg");
  strcpy (person_ptr2->address, "Permutation City");
  person_ptr2->year_of_birth = 1961;
  /* Don't know his birthday */
  puts ("Data contained:");
  puts (person1.name);
  puts (person2.name);
  puts (person_ptr1->name);
  puts (person_ptr2->name);

  return 0;
}
```

Any trailing items not initialized by data you specify are set to zero.

20.2 Memory allocation

Most variables in C have a fixed size. For example, a string declared to be 200 bytes long will always be 200 bytes long throughout the program. Sometimes, however, you will need variables whose size can vary. You might want a string whose size can vary between 0 and 100 kilobytes, for instance. We have already seen occasions where this sort of string is needed with the `getline` function. (See Section 16.2.6.1 [getline], page 127.)

This is where *dynamic data*, or data whose size can vary, comes in. Dynamic data is created via the process of *memory allocation*, that is, assigning a block of memory to a variable. Blocks of memory are usually assigned with the `malloc` function (the function name is from the phrase "memory allocation"), and can be resized with the `realloc` ("memory reallocation") function, and even merged back into the pool of available memory with the `free` function.

The `malloc` function takes one argument, the number of bytes to allocate. It returns a void pointer, which provides the address of the beginning of a block of memory that the program can use. This void pointer can be assigned to any other type of pointer. The only way to make use of the block of memory that has been allocated is through its pointer; in that sense, the block is not a "real" variable, that is to say, you cannot assign a value to the memory block directly. Instead, the address returned by `malloc` enables you to use the block indirectly; in this way, the block can contain any kind of value a real variable can. Having to use blocks indirectly through pointers is a small price to pay for the flexibility of dynamic data.

The following code example allocates a ten-byte string:

```
char *my_string;
my_string = (char *) malloc(10+1);
```

Notice that the void pointer returned by `malloc` is cast to a character pointer (type `char *`) before it is assigned to `my_string`. (See Section 5.4 [The cast operator], page 22.) Also notice that we have actually allocated 11 bytes of space; this is because the 11th byte must contain a null character that terminates the string but does not count toward its actual length. **Careful!** The newly-allocated block will be filled with garbage.

To reallocate the memory, use the `realloc` function. This function takes two parameters. The first is the pointer to the memory block to be reallocated, and the second is a number of type `size_t` that specifies the new size for the block. It returns a void pointer to the newly reallocated block. Here is how to reallocate the block allocated for `my_string` above, to a new size of 1000 bytes:

```
my_string = (char *) realloc (my_string, 1001);
```

The new block will contain all the data in the old block, followed by enough space to pad out the block to the new length. The new space will be filled with garbage.

Finally, to free up the memory allocated to a block and return it to the common pool of memory available to your program, use the `free` function, which takes only one argument, the pointer to the block you wish to free. It does not return a value.

```
free (my_string);
```

It is also possible to allocate the memory for a structure when it is needed and use the '`->`' operator to access the members of the structure, since we must access the structure via a pointer. (See the code sample following the next paragraph for an example of how to do this.) If you are creating complex data structures that require hundreds or thousands of structure variables (or more), the ability to create and destroy them dynamically can mean quite a savings in memory.

It's easy enough to allocate a block of memory when you know you want 1000 bytes for a string, but how do you know how much memory to allocate for a structure? For this task, C provides the `sizeof` function, which calculates the size of an object. For example, `sizeof (int)` returns the numbers of bytes occupied by an integer variable. Similarly, `sizeof (struct personal_data)` returns the number of bytes occupied by our `personal_data` structure. To allocate a pointer to one of these structures, then set the `year_of_birth` member to 1852, you would write something like the following:

```
struct personal_data* my_struct_ptr;

my_struct_ptr = (struct personal_data*)
  malloc (sizeof (struct personal_data));
my_struct_ptr->year_of_birth = 1852;
```

20.3 `union`

A union is like a structure in which all of the members are stored at the same address. Only one member can be in a union at one time. The `union` data type was invented to prevent the computer from breaking its memory up into many inefficiently sized chunks, a condition that is called *memory fragmentation*.

The `union` data type prevents fragmentation by creating a standard size for certain data. When the computer allocates memory for a program, it usually does so in one large block of bytes. Every variable allocated when the program runs occupies a segment of that block. When a variable is freed, it leaves a "hole" in the block allocated for the program. If this hole is of an unusual size, the computer may have difficulty allocating another variable to "fill" the hole, thus leading to inefficient memory usage. Since unions have a standard data size, however, any "hole" left in memory by freeing a union can be filled by another instance of the same type of union. A union works because the space allocated for it is the space taken by its largest member; thus, the small-scale memory inefficiency of allocating space for the worst case leads to memory efficiency on a larger scale.

20.3.1 Declaration of unions

A union is declared in the same way as a structure. It has a list of members, as in the example below:

```
union int_or_float
{
  int int_member;
  float float_member;
};
```

Declaring union variables is similar to declaring structure variables:

```
union int_or_float my_union1, my_union2;
```

Just like structures, the members of unions can be accessed with the '`.`' and '`->`' operators. However, unlike structures, the variables `my_union1` and `my_union2` above can be treated as either integers or floating-point variables at different times during the program. For example, if you write `my_union1.int_member = 5;`, then the program sees `my_union1` as being an integer. (This is only a manner of speaking. However, `my_union1` by itself does not have a value; only its members have values.) On the other hand, if you then type `my_union1.float_member = 7.7;`, the `my_union` variable loses its integer value. It is crucial to remember that a union variable can only have one type at the same time.

20.3.2 Using unions

One way to tell what type of member is currently stored in the union is to maintain a flag variable for each union. This can be done easily with enumerated data. For example, for the `int_or_float` type, we might want an associated enumerated type like this:

```
enum which_member
{
  INT,
  FLOAT
};
```

Notice that we used all-uppercase letters for the enumerated values. We would have received a syntax error if we had actually used the C keywords `int` and `float`.

Associated union and enumerated variables can now be declared in pairs:

```
union int_or_float my_union1;
enum which_member my_union_status1;
```

Handling union members is now straightforward. For example:

```
switch (my_union_status1)
  {
  case INT:
    my_union1.int_member += 5;
    break;
  case FLOAT:
    my_union1.float_member += 23.222333;
    break;
  }
```

These variables could even be grouped into a structure for ease of use:

```
struct multitype
{
    union int_or_float number;
    enum which_member status;
};

struct multitype my_multi;
```

You would then make assignments to the members of this structure in pairs:

```
my_multi.number.int_member = 5;
my_multi.status = INT;
```

20.4 Complex data structures

When building data structures, it is best to model the situation in question clearly and efficiently. Different types of data structure are good for different things. For example, arrays are good for storing tabular information. A chessboard looks like a two-dimensional array, so such an array is a good data structure to model a chess game. In this section we will examine more complex data structures that are useful for modeling more complex situations.

20.4.1 Data structure diagrams

Sometimes you will want to draw a picture that shows how to solve a problem by displaying how all its parts are connected. Such a picture is called a *structure diagram*.

Consider a hypothetical application that stores a map of the local countryside. This program must store information about individual towns and be able to give directions to the user about how to get from one town to another. A person driving a car from town to town might use a map, but the application programmer might use a structure diagram. Here is a structure diagram for the imaginary town of New Haven, and its neighboring towns North Haven, East Haven, South Haven, and West Haven:

```
                North Haven

                     ^
                     |
                     v

West Haven <-> New Haven <-> East Haven

                     ^
                     |
                     v

                South Haven
```

Once you have a structure diagram that represents your information, you can create a data structure that translates the structure diagram into the computer's memory. In this case, we can create a "town structure" that contains pointers to the towns that lie at the end of roads in the various compass directions. The town structure might look something like this:

```
struct town
{
  struct town *north;
  struct town *south;
  struct town *east;
  struct town *west;
  char name[50];
};
```

If the user of this hypothetical application wishes to know what is to the north of a particular town, the program only has to check that town's `north` pointer.

20.4.2 Dynamic data structures, Pointers and Dynamic Memory

For programs dealing with large sets of data, it would be a nuisance to have to name every structure variable containing every piece of data in the code — for one thing, it would be inconvenient to enter new data at run time because you would have to know the name of the variable in which to store the data when you wrote the program. For another thing, variables with names are permanent — they cannot be freed and their memory reallocated, so you might have to allocate an impractically large block of memory for your program at compile time, even though you might need to store much of the data you entered at run time temporarily.

Fortunately, complex data structures are built out of dynamically allocated memory, which does not have these limitations. All your program needs to do is keep track of a pointer to a dynamically allocated block, and it will always be able to find the block.

A complex data structure is usually built out of the following components:

nodes Dynamically-allocated blocks of data, usually structures.

links Pointers from nodes to their related nodes.

root The node where a data structure starts, also known as the *root node*. The address of the root of a data structure must be stored explicitly in a C variable, or else you will lose track of it.

There are some advantages to the use of dynamic storage for data structures:

- As mentioned above, since memory is allocated as needed, we don't need to declare how much we shall use in advance.
- Complex data structures can be made up of lots of "lesser" data structures in a modular way, making them easier to program.
- Using pointers to connect structures means that they can be re-connected in different ways as the need arises. (Data structures can be sorted, for example.)

20.4.3 Lists and trees

Two data structures that use nodes and links are very common: the linked list and the binary tree

20.4.3.1 Linked lists.

A *linked list* is a linear sequence of structures joined together by pointers. Each node's pointer links to the next node in the sequence. Linked lists have two main advantages over one dimensional arrays: they can be sorted easily simply by redirecting pointers, and they can be made any length at all dynamically.

Here is an example of a structure type from a linked list:

```
struct list_node
{
  double value;
  struct list_node *next;
};
```

Here the `value` member holds the actual content of the node, in this case a double-precision floating-point number, and the `next` member points to the next node in the list.

You will often encounter another basic kind of linked list, called a *doubly-linked list*. Each node in a doubly-linked list contains not only a pointer to the next node, but also to the previous node. This kind of list makes it easier to determine what the node preceding a node is, as well as the node succeeding it.

20.4.3.2 Binary trees

A *binary tree* is a data structure in which each node contains links to two successor nodes, so that the whole structure is shaped like a branching tree. A typical use for a binary tree might be storing genealogical information; since (at this point in human evolution) every individual has two parents, each node can represent a person and the two linked nodes can represent that person's mother and father. Let's extend our `personal_data` structure to incorporate this kind of information:

```
struct personal_data
{
  char name[100];
  char address[200];
  int year_of_birth;
  int month_of_birth;
  int day_of_birth;

  struct personal_data *mother;
  struct personal_data *father;
};
```

20.4.4 Setting up a data structure

Plan your data structures well, before you write any program code. Changes in program code may not affect data structures, but changes to data structures will likely imply drastic changes to program code.

20.4.4.1 Designing your data structure

The steps you should take in designing a data structure follow a basic pattern:

1. Group together all the kinds of information that must be stored and define a structure type with a member for each kind of information.
2. Add structure pointers to the structure type to reflect the way in which each bundle of information is connected to the others.
3. Design the algorithms to handle the memory allocation, node linking, and data storage.

20.4.4.2 Initializing your data structure

Once you understand your data structure, you can set about initializing it in the following way:

1. Declare your structure type. For example:
   ```
   struct town
   {
     struct town *north;
     struct town *south;
     struct town *east;
     struct town *west;
     char name[50];
   };
   ```
2. Declare two pointers to this type:
   ```
   struct town *root, *current;
   ```
   The `root` pointer is used to point to the root node of the data structure, and the `current` pointer points to the node with which we are currently working.
3. Allocate memory for the root node:

```
root = (struct town *) malloc (sizeof (struct town));
```

Be sure to check for errors. The variable `root` will be a null pointer if no memory could be allocated for the node.

4. Initialize the members of the root node:

```
root->north = NULL;
root->south = NULL;
root->east  = NULL;
root->west  = NULL;
strcpy (root->name, "New Haven");
```

Note that `NULL` pointers tell the program when it has come to the edge of the data structure, that is, when it has found a link that doesn't lead anywhere. At the moment, the links of the root node do not point anywhere. This will change as we add more nodes to the data structure.

5. Create a new, non-root node:

```
current = (struct town *) malloc (sizeof (struct town));
```

6. Initialize the current node:

```
current->north = NULL;
current->south = root;
current->east  = NULL;
current->west  = NULL;
strcpy (current->name, "North Haven");
```

7. Link neighboring nodes to the current node, as appropriate:

```
root->north = current;
```

8. Repeat steps 5 through 7, as necessary.

See Section 21.3 [Controlled recursion with data structures], page 216, for a practical example of building a simple linked list programmatically.

20.5 Further data structure examples

See Chapter 24 [Example programs], page 231, to see a complete application program that uses a complex data structure with nodes and links to stores its data later in the book.

20.6 Questions 20

1. What is the difference between a structure and a union?
2. What is a member?
3. If `foo` is a structure variable, how would you find out the value of its member `bar`?
4. If `foo` is a pointer to a structure variable, how would you find out the value of its member `bar`?
5. How are data usually linked to make a complex data structure?
6. Every structure variable in a complex data structure must have its own variable name. True or false?
7. How are the members of structures accessed in a data structure?

8. Write a small program to make linked list that contains three nodes long and set all their values to be zero. Can you automate this program with a loop? Can you make it work for any number of nodes?

21 Recursion

The program that swallowed its tail.

This chapter is about functions that call themselves. Consider the program below:

```
#include <stdio.h>

void black_hole()
{
  black_hole();
}

/* To shorten example, not using argp */
int main ()
{
  black_hole();
  return 0;
}
```

The `main` function calls the `black_hole` function, which calls itself, which calls itself, which calls... Once the control flow enters `black_hole`, it will never exit. This kind of function is called a *recursive function*, and a function's act of calling itself is called *recursion*.

21.1 The stack

What happens when we run the last example program? The `black_hole` function calls itself indefinitely. Each function call uses up a small portion of the computer's memory called the *stack*. Eventually all of this memory is used up, and a kind of error called a *stack overflow* occurs. The program then crashes with a *Segmentation fault* error.

It is sometimes helpful to think of a function as a robot that does a job. A function definition in effect provides the blueprints for a robot. When the function is executed, it is as though a robot is built on an assembly line in a robot factory. A recursive function is like a robot that builds a copy of itself on the same assembly line. The second robot is identical to the first in every way, except that it is an assistant to the first robot, and has been passed different arguments. This second robot may in turn build a copy of itself as well, and so on. It is crucial that the process of robots building robots stop at some point; otherwise, the robot factory will run out of raw materials (that is, computer memory), and the assembly line will grind to a halt.

21.1.1 The stack in detail

Let's examine this process in detail. When one function calls another function in a C program, control passes from the first function to the second function. When the second function ends, control passes back to the statement in the first function that immediately follows the function call. But how does the computer know where in its memory this statement resides?

The answer is simple. The computer keeps a list of the addresses in memory of the places to which it must return, no matter how many function calls are made. This list is the stack.

The stack gets its name from the fact that it is a *LIFO*, or *last in, first out* structure, meaning that the last item to be *pushed onto* the stack is the first item to be *popped off*. It works, in other words, like the stack of dinner plates you keep in your kitchen cabinet. As you wash plates, you pile them one by one on top of the stack, and when you want a plate, you take one from the top of the stack. The stack of plates in your cabinet is therefore also a last in, first out structure, like the computer's stack.

When one C function calls a second function, the computer leaves itself an address at the top of the stack of where it should return when it has finished executing the second function. If the second function calls a third function, the computer will push another address onto the stack. When the third function has finished executing, the computer pops the top address off the stack, which tells it where in the second function it should return. When the second function has finished, the computer again pops the top address off the stack — which tells it where in the first function it should return. Perhaps the first function then calls another function, and the whole process starts again.

What happens when `black_hole` calls itself? The computer makes a note of the address it must return to and pushes that address onto the top of the stack. It begins executing `black_hole` again, and encounters another call to `black_hole`. The computer pushes another address onto the top of the stack, and begins executing `black_hole` again. Since the program has no chance of popping addresses off the stack, as the process continues, the stack gets filled up with addresses. Eventually, the stack fills up and the program crashes.

21.2 Controlled recursion

If that were all there is to recursion, no one would ever use it. However, recursion can be limited so it does not go out of control. Controlled recursion can be a powerful programming technique.

When we discussed data structures, we remarked that programs and data structures should aim to model the situation they deal with closely. Some structures, both in real life and in computer memory, are made up of many levels of detail, and the details are roughly the same at every level. For example, a genealogical tree starts with an individual with two parents, each of whom has two parents, each of whom... These sorts of structure are called *self-similar*.

Since recursion employs functions that contain calls to themselves, in effect creating multiple self-similar levels of detail, controlled recursion is useful for dealing with self-similar problems.

Recursive functions can be controlled by making sure that there is a safe way to exit them at some point in the chain of function calls. The number of times recursion takes place is limited by making a decision about whether the function calls itself or not. Simply put, somewhere along the chain of function calls, the function makes the decision not to call itself again, in a process nicknamed *bottoming out*. At that point, the program begins popping addresses off the stack and returning to the previous functions. Eventually, the very first function in the chain terminates, and the program ends successfully.

A standard example of controlled recursion is the factorial function. This is a mathematical function which is important in statistics. The factorial function is defined to be the product (multiplication) of all integers from 1 to the parameter of the function. (The factorial of 0 is 1.)

Here are some examples of the factorial function. These are not executable C code examples, but pseudocode:

```
factorial(3) == 1 * 2 * 3          ==   6
factorial(4) == 1 * 2 * 3 * 4      ==  24
factorial(3) == 1 * 2 * 3 * 4 * 5 == 120
```

Formally, the factorial function is defined by two equations. (Again, these are in pseudocode).

```
factorial(n) = n * factorial(n-1)
factorial(0) = 1
```

The first of these statements is recursive, because it defines the value of `factorial(n)` in terms of `factorial(n-1)`. The second statement allows the function to "bottom out".

Here is a short code example that incorporates a `factorial` function.

```
#include <stdio.h>

int factorial (int n)
{
  if (n == 0)
    return 1;
  else
    return (n * factorial (n-1));
}

/* To shorten example, not using argp */
int main ()
{
  printf ("%d\n", factorial(3));
  return 0;
}
```

Let's follow the control flow in this program to see how controlled recursion can work. The `main` function prints the value of `factorial(3)`. First, the `factorial` function is called with the parameter 3. The function tests whether its parameter `n` is zero. It is not, so it takes the `else` branch if the `if` statement, which instructs it to return the value of `factorial(3-1)`. It therefore calls itself recursively with a parameter of 2.

The new call checks whether its parameter is zero. It isn't (it's 2), so it takes the `else` branch again, and tries to calculate `2 * factorial (1)`. In order to do so, it calls itself recursively with a value of 2-1, or 1. The new call checks whether its parameter is zero. It is actually 1, so it takes the `else` branch again and attempts to calculate `1 * factorial (0)`. In order to do so, it calls itself again with the parameter 0.

Again, the function checks whether its parameter is zero. This time it is, so the function bottoms out. It takes the first branch of the `if` statement and returns a value of 1. Now the previous function call can also return a value, and so on, until the very first call to `factorial` terminates, and the function returns a value of 6.

To sum up, the expression `factorial(3)` goes through the following steps before finally being evaluated:

```
factorial (3) == 3 * factorial(2)
              == 3 * (2 * factorial(1))
```

```
== 3 * (2 * (1 * factorial(0)))
== 3 * (2 * (1 * 1)))
== 6
```

Note: Make sure that the test for whether to bottom out your recursive function does not depend on a global variable.

Suppose you have a global variable called `countdown`, which your recursive function decrements by 1 every time it is called. When `countdown` equals zero, your recursive function bottoms out. However, since other functions than the recursive function have access to global variables, it is possible that another function might independently change `countdown` in such a way that your recursive function would never bottom out — perhaps by continually incrementing it, or perhaps even by setting it to a negative number.

21.3 Controlled recursion with data structures

Self-similar data structures are sometimes called *recursive data structures*. The simplest recursive data structure is the linked list. At every node in a linked list, there is data of a certain type and a link to the next node. The next simplest recursive data structure is the binary tree, which splits into two branches at every node. Recursive functions be useful for manipulating such recursive data structures.

The following code example makes use of recursion to print the value contained in the last node in a linked list.

```
#include <stdio.h>

struct list_node
{
  int data;
  struct list_node *next;
};

struct list_node *last_node (struct list_node *node)
{
  if (node->next == NULL)
    return node;
  else
    return last_node (node->next);
}

/* To shorten example, not using argp */
int main ()
{
  struct list_node *root;
  struct list_node *current;
  struct list_node *old;
  struct list_node *last;

  /* Initialize list. */
  root = (struct list_node *) malloc (sizeof (struct list_node));
  root->data = 1;
  old = root;
```

```
    current = (struct list_node *) malloc (sizeof (struct list_node));
    current->data = 2;
    old->next = current;
    old = current;

    current = (struct list_node *) malloc (sizeof (struct list_node));
    current->data = 3;
    old->next = current;
    current->next = NULL;

    /* Print data in last node. */
    last = last_node (root);
    printf ("Data in last node is %d.\n", last->data);

    return 0;
  }
```

This example program prints out the following line:

```
Data in last node is 3.
```

The `last_node` function, when passed a pointer to a node (such as the root), follows the linked list to its end from that point, and returns a pointer to that node. It does so through recursion. When it is passed a pointer to a node, it checks whether that node's `next` link is a null pointer. If the pointer is null, `last_node` has found the last node, and bottoms out, returning a pointer to the current node; otherwise, it calls itself with a pointer to the next node as a parameter.

21.4 Recursion summary

Recursion can be a powerful programming technique, especially when dealing with mathematical functions such as factorialisation that lend themselves naturally to recursion, or with self-similar data structures. There is a major disadvantage to recursion, however, and that is the amount of memory required to make it work. Do not forget that the program stack grows each time a function call is made. If a function calls itself too many times, your program will run out of memory and crash. Recursive programming can also be difficult; runaway recursion is a common error. Therefore, be judicious in your use of recursion.

21.5 Questions 21

1. What is a recursive function?
2. What is a program stack, and what is it for?
3. State the major disadvantage of recursion.

22 Style

C has no rules about when to start new lines, where to place whitespace, and so on. Users are free to choose a style which best suits them, but unless a strict style is adopted, sloppy programs tend to result.

In older compilers, memory restrictions sometimes necessitated bizarre, cryptic styles in the interest of efficiency. However, contemporary compilers such as GCC have no such restrictions, and have optimizers that can produce faster code than most programmers could write themselves by hand, so there are no excuses not to write programs as clearly as possible.

No simple set of rules will ever provide a complete methodology for writing good programs. In the end, experience and good judgment are the factors which determine whether you will write good programs. Nevertheless, a few guidelines to good style can be stated.

Many of the guidelines in this chapter are the distilled wisdom of countless C programmers over the decades that C has existed, and some come directly from the *GNU Coding Standards*. That document contains more good advice than can be crammed into this short chapter, so if you plan to write programs for the Free Software Foundation, you are urged to consult section "Table of Contents" in *GNU Coding Standards*.

22.1 Formatting code

Place the open curly bracket that starts the body of a C function in the first column of your source file, and avoid placing any other open brackets or open parentheses in that column. This will help many code-processing utilities find the beginnings of your functions. Similarly, you should also place the name of your functions within your function definitions in the first column. Thus, your functions should resemble the following example:

```
static char *
concat (char *s1, char *s2)
{
  ...
}
```

When you split an expression into multiple lines, split it before an operator, not after one. Here is the right way:

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```

Don't declare multiple variables in one declaration that spans lines. Start a new declaration on each line instead. For example, instead of this:

```
int    foo,
       bar;
```

write either this:

```
int foo, bar;
```

or this:

```
int foo;
int bar;
```

22.2 Comments and style

Comments are crucial for other programmers trying to understand your code. Every program should start with a comment saying briefly what it is for. Example: '`fmt - filter for simple filling of text`'. Similarly, you should put a comment on each function saying what the function does, what sort of arguments it takes, what the possible values of arguments mean, and what they are used for.

Please write all comments in a GNU program in English, because English is the one language that nearly all programmers in all countries can read.

22.3 Variable and function names

The names of variables and functions in a program serve as comments of a sort, so try to give your variables descriptive names (for example, `num_of_books`, `cost_per_entry`, or `distance_from_center`). Names should be in English, like other comments.

Use underscores rather than internal capitalization in names, so that Emacs word commands can be useful within them — thus `distance_from_center` rather than `distanceFromCenter` or `DistanceFromCenter`. In fact, upper-case letters should be reserved for macros and `enum` constants. Macros should be completely in upper case, for example `STANDARD_SIZE`.

It used to be common practice to use the same local variables (with names like `temp`) over and over for different purposes within one function. Instead, it is better to declare a separate local variable for each distinct purpose, and give it a meaningful name. This not only makes programs easier to understand, it also facilitates optimization by good compilers.

22.4 Declarations and initialization

You should explicitly declare the types of all objects. For example, explicitly declare all arguments to functions, and declare all function that return integers to return type `int`, even though the ANSI Standard permits omitting the `int`.

If there are only a few declarations, then initializing variables where you declare them can be tidy, but if there are many variables to declare, then it is usually better to declare and initialize separately, for the sake of clarity. In a long function, it is often good to initialize the variable near where you are using it, so that someone reading the code does not have to hunt around in the function to discover its initial value. (See Section 5.3 [Initialization], page 22.)

22.5 Global variables and style

Global variables have caused almost as much controversy as the `goto` statement. Some programmers say you should never use them. Other programmers use them on a regular basis. In fact, while global variables should not be overused, they can simplify your code considerably. The following guidelines may help you decide where to use globals.

- Always think of using local variables first. Global variables can puncture the *encapsulization* of your functions, that is, the logical isolation of your functions from the rest of your code. It is difficult to see what variables are being passed to a function unless they are all passed as parameters, so it is easier to debug a program when encapsulization is maintained.

- Local variables may be impractical, however, if they mean passing the same dozen parameters to multiple functions; in such cases, global variables will often streamline your code.
- Data structures that are important to the whole program should be defined globally. In “real programs” such as GNU Emacs, there are far more global variables than there are local variables visible in any one function.

Finally, don’t use local variables or parameters that have the same names as global identifiers. This can make debugging very difficult.

22.6 Hidden operators and style

Hiding operators away inside other statements can certainly make programs *look* elegant and compact, but it can make programs harder to understand. Never forget that besides being a set of instructions to the computer, programming is a form of communication to other programmers. Be kind to the reader of your program. It could be you in months or years to come.

Statements such as:

```
if ((my_int = (int)my_char++) <= --my_int2)
  {
    ...
  }
```

are not good style, and are no more efficient than the more longwinded:

```
my_int = (int) my_char;
my_char++;
my_int2--;

if (my_int <= my_int2)
  {
    ...
  }
```

22.7 Final words on style

It is easy to support pre-ANSI-Standard compilers in most programs, so if you know how to do that and a program you are maintaining has such support, you should try to keep it working.

Whatever style you use, use it consistently. A mixture of styles within a single program tends to look ugly and be hard to read and maintain. If you are contributing changes to an existing program, it is best to follow the style of that program.

22.8 Questions 22

1. Where should the name of a program and the opening bracket of a function definition begin?
2. In what human language should comments be written for the GNU Project? Why?
3. Which is better as the name of a variable: `plotArea`, `PlotArea`, or `plot_area`? Why?
4. Why is it important to initialize a variable near where it is used in a long function?
5. Give an example of a case where using local variables is impractical.

23 Debugging

True artificial intelligence has not yet been achieved. C compilers are not intelligent, but unconscious: mechanical in the derogatory sense of the word. Therefore, debugging your programs can be a difficult process. A single typographical error can cause a compiler to completely misunderstand your code and generate a misleading error message. Sometimes a long string of compiler error messages are generated because of a single error in your code. To minimize the time you spend debugging, it is useful to become familiar with the most common compiler messages and their probable causes.

The first section in this chapter lists some of these common compile-time errors and what to do about them. The next two sections discuss run-time errors in general, and mathematical errors in particular. The final section introduces GDB, the GNU Debugger, and explains some simple steps you can take to debug your programs with it.

23.1 Compile-time errors

In this section, we will examine a variety of compile-time errors and what you can do about them. The aim is not to be a comprehensive guide to everything that can go wrong with your program and all the corresponding error messages, but rather to give you a taste of the kinds of errors you are likely to make, and to build your confidence by showing that even fairly scary-looking error messages often have a simple cause.

23.1.1 parse error at. . ., parse error before. . .

This is a general-purpose syntax error. It is frequently caused by a missing semicolon. For example, the following code:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  printf ("Hello, world!\n")
  return 0;
}
```

generates the following error:

```
semicolon.c: In function 'main':
semicolon.c:6: parse error before 'return'
```

Adding a semicolon (‘;’) at the end of the line `printf ("Hello, world!")` will get rid of this error.

Notice that the error refers to line 6, but the error is actually on the previous line. This is quite common. Since C compilers are lenient about where you place whitespace, the compiler treats line 5 and line 6 as a single line that reads as follows:

```
printf ("Hello, world!\n") return 0;
```

Of course this code makes no sense, and that is why the compiler complains.

Often a missing curly bracket will cause one of these errors. For example, the following code:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  if (1==1)
    {
      printf ("Hello, world!\n");


  return 0;
}
```

generates the following error:

```
brackets.c: In function ‘main’:
brackets.c:11: parse error at end of input
```

Because there is no closing curly bracket for the `if` statement, the compiler thinks the curly bracket that terminates the `main` function actually terminates the `if` statement. When it does not find a curly bracket on line 11 of the program to terminate the `main` function, it complains. One way to avoid this problem is to type both members of a matching pair of brackets before you fill them in.

23.1.2 undefined reference to...

This error is often generated because you have typed the name of a function or variable incorrectly. For example, the following code:

```
#include <stdio.h>

void print_hello()
{
  printf ("Hello!\n");
}

/* To shorten example, not using argp */
int main()
{
  Print_hello();
  return 0;
}
```

generates the following rather forbidding error:

```
/tmp/cc9KXhmV.o: In function ‘main’:
/tmp/cc9KXhmV.o(.text+0x1f): undefined reference to ‘Print_hello’
collect2: ld returned 1 exit status
```

The answer, however, is very simple. C is case-sensitive. The `main` function calls the function `Print_hello` (with a capital ‘`P`’), but the correct name of the function is `print_hello` (with a lower-case ‘`p`’). The linker could not find a function with the name `Print_hello`.

23.1.3 unterminated string or character constant

This error is often generated by code like the following:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  printf("hello!\n);
  printf("Hello again!\n");
  return 0;
}
```

The actual error message received was:

```
missquotes.c:6: unterminated string or character constant
missquotes.c:5: possible real start of unterminated constant
```

The compiler never found a close quote (‘"’) for the string ‘`Hello!\n`’. It read all the text up from the quote in the line `printf("Hello!\n);` to the *first* quote in the line `printf("Hello again!\n");` as a single string. Notice that GCC helpfully suggests that it is line 5 that actually contains the unterminated string. GCC is pretty smart as C compilers go.

23.2 ...undeclared (first use in this function)

This is similar to the ‘`undefined reference to...`’ error, but instead of referring to an undefined function, you are referring to an undefined variable.

Sometimes this is a scope problem. You might get this error if you tried to refer to another function's local variable. For example:

```
#include <stdio.h>

void set_value()
{
  int my_int = 5;
}

/* To shorten example, not using argp */
int main()
{
  my_int = 23;
  return 0;
}
```

The variable `my_int` is local to the function `set_value`, so referring to it from within `main` results in the following error:

```
undec.c: In function ‘main’:
undec.c:10: ‘my_int’ undeclared (first use in this function)
undec.c:10: (Each undeclared identifier is reported only once
undec.c:10: for each function it appears in.)
```

23.2.1 different type arg

You might get this warning if you mismatch a parameter to `printf` and a conversion specifier. For example, the following code:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  int my_int = 5;
  printf ("%f", my_int);
  return 0;
}
```

produces the folliwing warning:

```
wrongtype2.c: In function ‘main’:
wrongtype2.c:6: warning: double format, different type arg (arg 2)
```

The '`%f`' conversion specifier requires a floating-point argument, while `my_int` is an integer, so GCC complains.

Note: GCC is quite lenient about type mismatches and will usually coerce one type to another dynamically without complaining, for example when assigning a floating-point number to an integer. This extends to mismatched parameters and conversion specifiers — although you may receive odd results from `printf` and so on, the causes of which may not be obvious. Therefore, in order to generate this warning, the '`-Wall`' option of GCC was used. This option causes GCC to be especially sensitive to errors, and to complain about problems it usually ignores. You will often find the '`-Wall`' option to be useful in finding tricky problems. Here is the actual command line used to compile this program:

```
gcc -Wall -o wrong wrongtype2.c
```

23.2.2 too few parameters..., too many parameters...

Consider the following program:

```
#include <stdio.h>

void tweedledee (int a, int b, int c)
{
}

void tweedledum (int a, int b)
{
}
```

```
/* To shorten example, not using argp */
int main()
{

  tweedledee (1, 2);
  tweedledum (1, 2, 3);

  return 0;
}
```

The `tweedledee` function takes three parameters, but `main` passes it two, whereas the `tweedledum` function takes two parameters, but `main` passes it three. The result is a pair of straightforward error messages:

```
params.c: In function ‘main’:
params.c:14: too few arguments to function ‘tweedledee’
params.c:15: too many arguments to function ‘tweedledum’
```

This is one reason for the existence of function prototypes. Before the ANSI Standard, compilers did not complain about this kind of error. If you were working with a library of functions with which you were not familiar, and you passed one the wrong number of parameters, the error was sometimes difficult to track. Contemporary C compilers such as GCC that follow the standard make finding parameter mismatch errors simple.

23.3 Run-time errors

This section examines errors that cannot be caught by the compiler and are exhibited only when the program is run.

23.3.1 Confusion of = and ==

Consider the following program:

```
#include <stdio.h>

/* To shorten example, not using argp */
int main()
{
  int my_int = 0;

  if (my_int = 1)
    {
      printf ("Hello!\n");
    }

  return 0;
}
```

What will this program do? If you guessed that it will print ‘`Hello!`’, you are correct. The assignment operator (`=`) was used by mistake instead of the equality operator (`==`). What is being tested in the above `if` statement is not whether `my_int` has a value of 1 (which would be written `if my_int == 1`), but instead what the value is of the assignment statement `my_int = 1`. Since the value of an assignment statement is always the result of the assignment, and `my_int` is here being assigned a value of 1, the result is 1, which C considers to be equivalent to `TRUE`. Thus, the program prints out its greeting.

Even the best C programmers make this mistake from time to time, and tracking down an error like this can be maddening. Using the ‘`-Wall`’ option of GCC can help at least a little by giving you a warning like the following:

```
equals.c: In function ‘main’:
equals.c:7: warning: suggest parentheses around assignment used as truth value
```

23.3.2 Confusing `foo++` and `++foo`

In many cases, the forms `foo++` and `++foo` are identical. However, if they are hidden inside another statement, there can be a subtle difference. For example:

```
my_array [++my_index] = 0;
```

The code `++my_index` cause `my_index` to be incremented by 1 *before* the assignment takes place, whereas `my_index++` would have cause `my_index` to be incremented *after* the assignment takes place. Sometimes you'll want one and sometimes the other. If you find that your program is miscalculating by a difference of 1 (this is called an *off-by-one bug* and is quite common), a prefix or postfix ‘`++`’ could be the cause. The same holds for other prefix and postfix operators, such as ‘`--`’.

23.3.3 Unwarranted assumptions about storage

Do not assume that the size of a structure is the sum of the sizes of its parts. The two may differ for various reasons; for example, the operating system may be aligning variables with specific addresses within the data structure. Furthermore, the elements of an array may not even be next to one another in memory.

This kind of code is always safe:

```
int my_array[3];

my_array[0] = 0;
my_array[1] = 0;
my_array[2] = 0;
```

This kind of code is not:

```
int my_array[3];

*my_array = 0;
*(my_array + (1 * sizeof(int))) = 0;
*(my_array + (2 * sizeof(int))) = 0;
```

While it is true that the variable `my_array` used without its square brackets is a pointer to the first element of the array, you must not assume that you can simply calculate a pointer to the third element with code like the following:

```
my_array + 2 * sizeof(int);
```

Do something like this instead:

```
&(my_array[2]);
```

23.3.4 Array out of bounds

When you get or set the value of an element of an array, GCC does not check whether you are working within the bound of the array. In the worst case, this can lead to your program crashing (but probably nothing worse happening on a GNU system). See Section 14.1 [Array bounds], page 90, for more information on this error. See Section 23.5 [Introduction to GDB], page 230, for information on how you can check whether you are violating array bounds, using the GNU Debugger.

23.3.5 Uncoordinated output

You may occasionally experience strange effects when writing output to the screen, such as no output at all until the input is complete, or spontaneous bursts of output at seemingly random intervals. This sort of problem usually has to do with the way the output is buffered. The solution is usually to write a newline character ('`\n`') to the output when you are ready to display it, or to use a function like `fflush` to flush the buffer. (See Section 16.1.5 [Stream buffering], page 117, for more information.)

23.3.6 Global variables and recursion

Global variables and recursion usually do not mix. Make sure that the test for whether to "bottom out" your recursive function does not depend on a global variable. See Section 21.2 [Controlled recursion], page 214, for more information on why this is a bad thing.

23.4 Mathematical errors

Mathematical errors are a special kind of run-time error. They may not necessarily cause your program to crash, but they are likely to produce all sorts of strange results if you are doing some complex calculations in your program. Consider the following line of code:

```
root = sqrt (-1.0);
```

Readers with a smattering of mathematics will recognise that this code cannot give a sensible answer. The square root of -1 is a complex number called i. The number i is a so-called imaginary number, and cannot be represented by a floating-point value, which is what the `sqrt` function returns.

What happens in such a case? Two things:

1. The value returned is a special floating-point macro such as `NAN` (which means "not a number") or `INFINITY`.
2. More importantly from a debugging standpoint, a floating-point *exception* occurs. An exception is an error condition, and when a floating-point exception is *raised*, as the jargon goes, an error flag is set in the operating system, signifying what kind of exception it was (in other words, what kind of error caused the exception to be raised).

There are several kinds of floating-point exception:

`FE_INVALID`: The "Invalid Operation" exception. Raised if the operands are invalid for the given operation, for example, if you are trying to take the square root of a negative number, as above.

`FE_DIVBYZERO`: The "Division by Zero" exception. Raised when a finite, nonzero number is divided by zero.

`FE_OVERFLOW`: The "Overflow" exception. Raised when the result cannot be expressed as a finite value, for example when a finite, nonzero number is divided by zero. Whenever this exception is raised, the `FE_INEXACT` exception is also raised.

`FE_UNDERFLOW`: The "Underflow" exception. Raised when an intermediate result is too small to be calculated accurately, or when an operation's rounded result is too small to be *normalized*. Normalisation, roughly speaking, is the process of converting a number to scientific notation, such as converting 235 to 2.35e2, where the *mantissa*, or number to the left of the '`e`', must not be zero. See Section 5.1.2 [Floating point variables], page 20, for more information on scientific notation.)

`FE_INEXACT`: The "Inexact" exception. Raised if a rounded result is not exact, for example when calculating an irrational number such as the square root of 2.

You can test for these exceptions with the `fetestexcept` function, which takes one parameter, a bitwise OR'd list of the exception flags from the list above for which you are testing, and returns a nonzero value containing a bitwise OR'd list of the flags you passed it for the exceptions that actually occurred. You can also clear selected flags with the `feclearexcept` function, which accepts a bitwise-OR'd list of exception flags to clear, and returns zero if it was successful. (You can pass either of these function the macro `FE_ALL_EXCEPT`, which contains all of the floating-point exception flags OR'd together.)

In case this explanation is unclear, let's look at a practical example.

23.5 Introduction to GDB

Blah blah blah.

23.6 Questions 23

Spot the errors in the following:

Blah blah blah.

24 Example programs

The aim of this section is to provide a substantial example of C programming, using input from and output to disk, GNU-style long options, and the linked list data structure (including insertion, deletion, and sorting of nodes).

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

#define NAME_LEN 100
#define ADDR_LEN 500

const char *argp_program_version =
"bigex 1.0";

const char *argp_program_bug_address =
"<bug-gnu-utilsgnu.org>";


/* This structure is used by main to communicate with parse_opt. */
struct arguments
{
  char *args[1];            /* No arguments to this function */
  int verbose;              /* The -v flag */
  char *infile;             /* Argument for -i */
  char *outfile;            /* Argument for -o */
};


struct personal_data
{
  char name[NAME_LEN];
  char address[ADDR_LEN];
  struct personal_data *next;
};


/*
  OPTIONS.  Field 1 in ARGP.
  Order of fields: {NAME, KEY, ARG, FLAGS, DOC}.
*/
static struct argp_option options[] =
  {
    {"verbose", 'v', 0, 0, "Produce verbose output"},

    {"input",   'i', "INFILE",  0,
     "Read addresses from INFILE"},

    {"output",  'o', "OUTFILE", 0,
```

```
     "Output to OUTFILE instead of to standard output"},

    {0}
  };


/*
  PARSER. Field 2 in ARGP.
  Order of parameters: KEY, ARG, STATE.
*/
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'v':
      arguments->verbose = 1;
      break;
    case 'i':
      arguments->infile = arg;
      break;
    case 'o':
      arguments->outfile = arg;
      break;
    case ARGP_KEY_ARG:
      if (state->arg_num >= 1)
{
  argp_usage(state);
}
      arguments->args[state->arg_num] = arg;
      break;
    case ARGP_KEY_END:
      if (state->arg_num < 1)
{
  argp_usage (state);
}
      break;
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}


/*
  ARGS_DOC. Field 3 in ARGP.
  A description of the non-option command-line arguments
```

```
  that we accept.
*/
static char args_doc[] = "ARG";


/*
  DOC.  Field 4 in ARGP.
  Program documentation.
*/
static char doc[] =
"bigex -- Add ARG new names to an address book file.\vThe largest code example in the GNU C Tutorial.";


/*
  The ARGP structure itself.
*/
static struct argp argp = {options, parse_opt, args_doc, doc};


struct personal_data *
new_empty_node()
{
  struct personal_data *new_node;

  new_node =  (struct personal_data*)
    malloc (sizeof (struct personal_data));

  strcpy (new_node->name, "");
  strcpy (new_node->address, "");
  new_node->next = NULL;

  return new_node;
}


struct personal_data *
create_node()
{
  int bytes_read;
  int nbytes;

  struct personal_data *current_node;
  char *name;
  char *address;

  current_node = new_empty_node();

  puts ("Name?");
  nbytes = NAME_LEN;
```

```
  name = (char *) malloc (nbytes + 1);
  bytes_read = getline (&name, &nbytes, stdin);
  if (bytes_read == -1)
    {
      puts ("ERROR!");
    }
  else
    {
      strncpy (current_node->name, name, NAME_LEN);
      free (name);
    }

  puts ("Address?");
  nbytes = ADDR_LEN;
  address = (char *) malloc (nbytes + 1);
  bytes_read = getline (&address, &nbytes, stdin);
  if (bytes_read == -1)
    {
      puts ("ERROR!");
    }
  else
    {
      strncpy (current_node->address, address, ADDR_LEN);
      free (address);
    }

  printf("\n");
  return current_node;
}



struct personal_data *
find_end_node (struct personal_data *current_node)
{
  if (current_node->next == NULL)
    {
      return current_node;
    }
  else
    {
      return find_end_node (current_node->next);
    }
}


int
list_length (struct personal_data *root)
{
```

```
  struct personal_data *current_node;
  int count = 0;

  current_node = root;

  while (current_node->next != NULL)
    {
      current_node = current_node->next;
      count++;
    }
  return count;
}


struct personal_data *
find_node (struct personal_data *root,
   int node_wanted)
{
  struct personal_data *current_node;
  int index = 0;

  current_node = root;

  while ((index < node_wanted) && (current_node->next != NULL))
    {
      current_node = current_node->next;
      index++;
    }
  return current_node;
}



delete_node (struct personal_data *root,
     int location)
{
  struct personal_data *previous_node;
  struct personal_data *current_node;

  previous_node = find_node (root, location - 1);
  current_node = find_node (root, location);
  previous_node->next = current_node->next;
}



insert_node (struct personal_data *root,
     struct personal_data *new_node,
     int location)
```

```
{
  struct personal_data *temp_ptr;
  struct personal_data *previous_node;

  previous_node = find_node (root, location - 1);
  temp_ptr = previous_node->next;

  previous_node->next = new_node;
  new_node->next = temp_ptr;
}


swap_nodes (struct personal_data *root, int a, int b)
{
  int temp;
  struct personal_data *node_a;
  struct personal_data *node_b;
  struct personal_data *temp_node;

  if (a > b)
    {
      temp = a;
      a = b;
      b = temp;
    }

  node_b = find_node (root, b);
  delete_node (root, b);

  node_a = find_node (root, a);
  delete_node (root, a);

  insert_node (root, node_b, a);
  insert_node (root, node_a, b);
}


sort_list (struct personal_data *root)
{
  int i, j, list_len, diff;

  list_len = list_length (root);
  for (i=2; i<=list_len; i++)
    {
      j = i;
      while (strcmp ( (find_node(root, j))->name,
      (find_node(root, j-1))->name) < 0)
{
  swap_nodes (root, j, j-1);
```

```
  j--;
}
    }
}


print_node (struct personal_data *current_node,
    FILE *save_stream)
{
  fprintf (save_stream, "%s%s",
   current_node->name,
   current_node->address);
}


print_list (struct personal_data *current_node,
    FILE *save_stream)
{
  print_node (current_node, save_stream);

  if (current_node->next != NULL)
    {
      print_list (current_node->next, save_stream);
    }
}




struct personal_data *
read_node (FILE *instream)
{
  int bytes_read;
  int nbytes;

  struct personal_data *current_node;
  char *name;
  char *address;
  char *blankline;
  int read_err = 0;

  current_node = new_empty_node();

  nbytes = NAME_LEN;
  name = (char *) malloc (nbytes + 1);
  bytes_read = getline (&name, &nbytes, instream);
  if (bytes_read == -1)
    {
      read_err = 1;
```

```
    }
  else
    {
      puts (name);
      strncpy (current_node->name, name, NAME_LEN);
      free (name);
    }

  nbytes = ADDR_LEN;
  address = (char *) malloc (nbytes + 1);
  bytes_read = getline (&address, &nbytes, instream);
  if (bytes_read == -1)
    {
      read_err = 1;
    }
  else
    {
      puts (address);
      strncpy (current_node->address, address, ADDR_LEN);
      free (address);
    }

  if (read_err)
    {
      return NULL;
    }
  else
    {
      return current_node;
    }
}


struct personal_data *
read_file (char *infile)
{
  FILE *input_stream = NULL;
  struct personal_data *root;
  struct personal_data *end_node;
  struct personal_data *current_node;

  root = new_empty_node();
  end_node = root;

  input_stream = fopen (infile, "r");
  if (input_stream)
    {
      while (current_node = read_node (input_stream))
{
```

```
  end_node->next = current_node;
  end_node = current_node;
  end_node->next = NULL;
}
    }
  return root;
}



/*
  The main function.
  Notice how now the only function call needed to process
  all command-line options and arguments nicely
  is argp_parse.
*/
int
main (int argc, char **argv)
{
  struct arguments arguments;
  struct personal_data *root;
  struct personal_data *end_node;
  struct personal_data *current_node;
  int i, newnum;
  FILE *save_stream;


  /* Set argument defaults */
  arguments.infile = NULL;
  arguments.outfile = NULL;
  arguments.verbose = 0;

  /* Where the magic happens */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  if (arguments.infile)
    {
      root = read_file (arguments.infile);
      end_node = find_end_node (root);
    }
  else
    {
      root = new_empty_node();
      end_node = root;
    }

  /* Where do we send output? */
  if (arguments.outfile)
    save_stream = fopen (arguments.outfile, "w");
```

```
  else
    save_stream = stdout;

  newnum = atoi (arguments.args[0]);

  for (i = 1; i <= newnum; i++)
    {
      current_node = create_node();
      end_node->next = current_node;
      end_node = current_node;
      end_node->next = NULL;
    }

  sort_list (root);
  print_list (root->next, save_stream);

  /* Close stream; skip error-checking for brevity of example */
  fclose (save_stream);

  /* If in verbose mode, print song stanza */
  if (arguments.verbose)
    {};

  return 0;
}
```

Appendix A A note from the original author

This book began life in 1987 as one of the early books on C programming. I wrote it during a summer vacation from University, in England in 1987. It was published by Dabs Press, a small publishing house which specialized in books for microcomputers, particularly Acorn's classic BBC micro. With the arrival of the Amiga, I was able to obtain a C compiler. I had had my eye on C for some time, and I felt at the time, that it was the best language I had seen to date for system programming. The publisher and I decided that C would become the 'BASIC' of the 16-bit microcomputer world, which it did. C took off, and the book sold very well for a number of years. As the contract expired, the book was lost in my files, until I met Richard Stallman and he asked me if I would give the tutorial to GNU. I agreed to recover the original files from diskettes and partly re-work them, to remove the emphasis from micro-computers and over to GNU. The result of that work was the first version of the tutorial. Having handed over the merchandise, I agreed that it would be a good thing for others to update and improve the tutorial. My only requirement was that I would be allowed to explain a few changes for which I would not want to be blamed. I cannot insist that such changes will not be made, but I can at least distance myself from them. They are fairly picky and even silly things, but for one reason or another, they mean a lot to me. The first has to do with grammar. The grammar which is written and printed in books today is often incorrect. Many colloquialisms and vernacular perversions of grammar are printed and believed to be correct. I am fairly sure that no such errors are mine! The other thing has to do with the style and formatting of C code. The placement of curly braces is something about which I agree with only a handful of people on the planet. Kernighan and Ritchie's original placement of curly braces is so horrendous that I would go so far as to call it "wrong", logically and aesthetically. The GNU indentation, which positions braces of the same level in straight vertical alignment, is much better, but in my opinion it gets its indentation wrong. I would indent 3 positions before the first brace, and keep the text within braces aligned with the braces themselves, rather than indenting within the braces, as GNU does. That, in my personal opinion, makes it easier to identify text and braces belonging together, and leads to optimal clarity. I also insist that curly braces be used around single statements, in loops and tests, even when not strictly required by the language. Finally, having grown up in England and lived in Norway, which have contradictory punctuation rules, I am utterly confused about punctuation and have probably made many errors. With that little spiel said, I now pass the torch to future authors and wish everyone luck. I am happy to see an old summer job not go to waste.

Mark Burgess, Oslo March 2001

Appendix B Reserved words in C

Blah blah blah.

Here is a list of all the reserved words in C. The set of reserved words above is used to build up the basic instructions of C; you can not use them in programs your write

Please note that this list is somewhat misleading. Many more words are out of bounds. This is because most of the facilities which C offers are in libraries that are included in programs. Once a library has been included in a program, its functions are defined and you cannot use their names yourself.

C requires all of these reserved words to be in lower case. (This does mean that, typed in upper case, the reserved words could be used as variable names, but this is not recommended.)

(A "d" by the word implies that it is used as part of a declaration.)

```
auto d                    if
     break                         int d
     case                          long d
     char d                   register d
     continue                      return
     default                       short d
     do                            sizeof
     double d                 static d
     else                          struct
     entry                         switch
     extern d                 typedef d
     float d                  union d
     for                           unsigned d
     goto                          while
```

also in modern implementations:

```
enum d
void d

const d
signed d
volatile d
```

Appendix C Precedence of operators

The highest priority operators are listed first.

Operator	*Operation*	*Evaluated*
`()`	`parentheses`	`left to right`
`[]`	`square brackets`	`left to right`
`++`	`increment`	`right to left`
`--`	`decrement`	`right to left`
`(type)`	`cast operator`	`right to left`
`*`	`the contents of`	`right to left`
`&`	`the address of`	`right to left`
`-`	`unary minus`	`right to left`
`~`	`one's complement`	`right to left`
`!`	`logical NOT`	`right to left`
`*`	`multiply`	`left to right`
`/`	`divide`	`left to right`
`%`	`remainder (MOD)`	`left to right`
`+`	`add`	`left to right`
`-`	`subtract`	`left to right`
`>>`	`shift right`	`left to right`
`<<`	`shift left`	`left to right`
`>`	`is greater than`	`left to right`
`>=`	`greater than or equal to`	`left to right`
`<=`	`less than or equal to`	`left to right`
`<`	`less than`	`left to right`
`==`	`is equal to`	`left to right`
`!=`	`is not equal to`	`left to right`
`&`	`bitwise AND`	`left to right`
`^`	`bitwise exclusive OR`	`left to right`
`|`	`bitwsie includive OR`	`left to right`
`&&`	`logical AND`	`left to right`
`||`	`logical OR`	`left to right`
`=`	`assign`	`right to left`
`+=`	`add assign`	`right to left`
`-=`	`subtract assign`	`right to left`
`*=`	`multiply assign`	`right to left`
`/=`	`divide assign`	`right to left`
`%=`	`remainder assign`	`right to left`
`>>=`	`right shift assign`	`right to left`
`<<=`	`left shift assign`	`right to left`

`&=`	`AND assign`	`right to left`
`^=`	`exclusive OR assign`	`right to left`
`|=`	`inclusive OR assign`	`right to left`

Appendix D Special characters

Control characters are invisible on the screen. They have special purposes usually to do with cursor movement and are written into an ordinary string or character by typing a backslash character \ followed by some other character. These characters are listed below.

A character can be any ASCII character, printable or not printable from values -128 to 127. (But only 0 to 127 are used.) Control characters i.e. non printable characters are put into programs by using a backslash \ and a special character or number. The characters and their meanings are:

'`\b`' backspace BS

'`\f`' form feed FF (also clear screen)

'`\n`' new line NL (like pressing return)

'`\r`' carriage return CR (cursor to start of line)

'`\t`' horizontal tab HT

'`\v`' vertical tab (not all versions)

'`\x`' ???

'`\"`' double quotes (not all versions)

'`\'`' single quote character '

'`\\`' backslash character \

'*ddd*' character *ddd* where ddd is an ASCII code given in octal or base 8. (See Appendix C)

Here is a code example that prints special characters:

```
/***************************************************/
/*                                                 */
/* Special Characters                              */
/*                                                 */
/***************************************************/

#include <stdio.h>

main ()

{
printf ("Beep! \7 \n");
printf ("ch = \'a\' \n");
printf (" <- Start of this line!! \r");
}
```

The output of this program is:

```
Beep!   (and the BELL sound)
ch = 'a'
 <- Start of this line!!
```

and the text cursor is left where the arrow points.

Appendix E Character conversion table

This table lists the decimal, octal, and hexadecimal numbers for characters 0 – 127.

Decimal	Octal	Hexadecimal	Character
0	0	0	CTRL-@
1	1	1	CTRL-A
2	2	2	CTRL-B
3	3	3	CTRL-C
4	4	4	CTRL-D
5	5	5	CTRL-E

6	6	6	CTRL-F
7	7	7	CTRL-G
8	10	8	CTRL-H
9	11	9	CTRL-I
10	12	A	CTRL-J
11	13	B	CTRL-K
12	14	C	CTRL-L
13	15	D	CTRL-M
14	16	E	CTRL-N
15	17	F	CTRL-O
16	20	10	CTRL-P
17	21	11	CTRL-Q
18	22	12	CTRL-R
19	23	13	CTRL-S
20	24	14	CTRL-T
21	25	15	CTRL-U
22	26	16	CTRL-V
23	27	17	CTRL-W
24	30	18	CTRL-X
25	31	19	CTRL-Y
26	32	1A	CTRL-Z
27	33	1B	CTRL-[
28	34	1C	CTRL-\
29	35	1D	CTRL-]
30	36	1E	CTRL-^
31	37	1F	CTRL-_
32	40	20	
33	41	21	!
34	42	22	"
35	43	23	#
36	44	24	$
37	45	25	%
38	46	26	&
39	47	27	’
40	50	28	(
41	51	29	)
42	52	2A	*
43	53	2B	+
44	54	2C	,
45	55	2D	-
46	56	2E	.
47	57	2F	/
48	60	30	0
49	61	31	1
50	62	32	2
51	63	33	3
52	64	34	4
53	65	35	5
54	66	36	6
55	67	37	7
56	70	38	8
57	71	39	9
58	72	3A	:
59	73	3B	;
60	74	3C	<
61	75	3D	=
62	76	3E	>
63	77	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	‘
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

Appendix F A word about `goto`

This word is redundant in C and encourages poor programming style. For this reason it has been ignored in this book. For completeness, and for those who insist on using it (may their programs recover gracefully) the form of the goto statement is as follows:

```
goto label;
```

`label` is an identifier which occurs somewhere else in the given function and is defined as a label by using the colon:

```
label : printf ("Ugh!  You used a goto!");
```

Appendix G Answers to questions

Blah blah blah.

Bibliography

Blah blah blah.

Glossary

Blah blah blah.

Code index

|

~

+

>

^

<

0

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Concept index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X

Z

Bits and pieces

This section is for random chunks of text that are too good to drop from the book, but were out-of-place where they were.

Allocating memory for strings

Neither of the methods above is any good if a program is going to be fetching a lot of strings from a user. It just isn't practical to define lots of static strings and expect the user to type into the right size boxes! The next step in string handling is therefore to allocate memory for strings personally: in other words to be able to say how much storage is needed for a string while a program is running. C has special memory allocation functions which can do this, not only for strings but for any kind of object. Suppose then that a program is going to get ten strings from the user. Here is one way in which it could be done:

1. Define one large, static string (or array) for getting one string at a time. Call this a string buffer, or waiting place.
2. Define an array of ten pointers to characters, so that the strings can be recalled easily.
3. Find out how long the string in the string buffer is.
4. Allocate memory for the string.
5. Copy the string from the buffer to the new storage and place a pointer to it in the array of pointers for reference.
6. Release the memory when it is finished with.

Characters

In C, single characters are written enclosed by single quotes. This is in contrast to strings of characters, which use double quotes ('`"..."`').

```
int ch;
ch = 'a';
```

would give `ch` the value of the character '`a`'. The same effect can also be achieved by writing:

```
char ch = 'a';
```

It is also possible to have the type:

```
unsigned char
```

This admits ASCII values from 0 to 255, rather than -128 to 127.

Assigning variables to one another

Not only can you assign numbers to variables, you can assign other variables to variables:

```
var1 = 23;
var2 = var1;
```

The variable or value on either side of the '`=`' symbol must usually be of the same type. However, integers and characters will interconvert because characters are stored by their ASCII codes (which are integers!) Thus the following will work:

```
int i;
char ch = 'A';

i = ch;

printf ("The ASCII code of %c is %d",ch,i);
```

The result of this would be:

```
The ASCII code of A is 65
```

Function pointers

You can create pointers to functions as well as to variables. Function pointers can be tricky, however, and caution is advised in using them.

Function pointers allow you to pass functions as a parameters to another function. This enables you to give the latter function a choice of functions to call. That is, you can plug in a new function in place of an old one simply by passing a different parameter. This technique is sometimes called *indirection* or *vectoring*.

To pass a pointer for one function to a second function, simply use the name of the first function, as long as there is no variable with the same name. Do not include the first function's parentheses or parameters when you pass its name.

For example, the following code passes a pointer for the function named `fred_function` to the function `barbara_function`:

```
void fred();
barbara (fred);
```

Notice that `fred` is declared with a regular function prototype before `barbara` calls it. You must also declare `barbara`, of course:

```
void barbara (void (*function_ptr)() );
```

Notice the parentheses around `function_ptr` and the parentheses after it. As far as `barbara` is concerned, any function passed to it is named `(*function_ptr)()`, and this is how `fred` is called in the example below:

```
#include <stdio.h>

void fred();
void barbara ( void (*function_ptr)() );
int main();

int main()
{
  barbara (fred);
  return 0;
}

void fred()
{
  printf("fred here!\n");
}
```

```
void barbara ( void (*function_ptr)() )
{
  /* Call fred */
  (*function_ptr)();
}
```

The output from this example is simply '`fred here!`'.

Again, notice how `barbara` called `fred`. Given a pointer to a function, the syntax for calling the function is as follows:

variable `=` `(*`*function_pointer*`)(`*parameter_list*`);`

For example, in the program below, the function `do_math` calls the functions `add` and `subtract` with the following line:

```
result = (*math_fn_ptr) (num1, num2);
```

Here is the example program:

```
#include <stdio.h>

int add (int, int);
int subtract (int, int);
int do_math (int (*math_fn_ptr) (int, int), int, int);
int main();

int main()
{
  int result;

  result = do_math (add, 10, 5);
  printf ("Addition = %d.\n", result);

  result = do_math (subtract, 40, 5);
  printf ("Subtraction = %d.\n\n", result);

  return 0;
}

int add (int num1, int num2)
{
  return (num1 + num2);
}


int subtract (int num1, int num2)
{
  return (num1 - num2);
}


int do_math (int (*math_fn_ptr) (int, int), int num1, int num2)
```

```
{
  int result;

  printf ("\ndo_math here.\n");

  /* Call one of the math functions passed to us:
     either add or subtract. */

  result = (*math_fn_ptr) (num1, num2);
  return result;
}
```

The output from this program reads:

```
do_math here.
Addition = 15.

do_math here.
Subtraction = 35.
```

You can also initialize a function pointer by setting it to the name of a function, then treating the function pointer as an ordinary function, as in the next example:

```
#include <stdio.h>

int main();
void print_it();
void (*fn_ptr)();

int main()
{
  void (*fn_ptr)() = print_it;

  (*fn_ptr)();

  return 0;
}

void print_it()
{
  printf("We are here!  We are here!\n\n");
}
```

Remember to initialize any function pointers you use this way! If you do not, your program will probably crash, because the uninitialized function pointer will contain garbage.

Table of Contents