THE LITTLE BOOK OF C

by Huw Collingbourne
(Bitwise Courses)

http://bitwisecourses.com/

written by
Huw Collingbourne

First edition: October 2013
Revision 1.1. July 2017

Introduction

In this short eBook I will explain the essential features of the C programming language. I will begin with the basic building blocks, so even if you've never programmed before, don't worry. I'll explain everything you need to know in order to understand how C programs are written and what they do when they are run.

If you already have some programming experience, say in a language such as C#, Ruby, Python or Java, you may want to skip the introductory lessons.

However, don't be too eager to skip too much in order to move onto the later sections of the course. While C has much in common with some other languages, such as Java, that have adopted its syntax, it also has a great many important differences. For example, C is not object oriented, it has no string data type and it is much 'lower level' (closer to the way the computer actually works) than many other languages. As a consequence, it doesn't offer as much protection from many errors.

In this course, I will highlight many of the common problems that novice C programmers make – everything from 'buffer overruns' to memory leaks. I'll also explain the special features of C such as its operators, *typedefs*, header files and loops.

Chapter 1 – Getting Started

This course comprises numerous video tutorials, sample C code to download and try out, plus this eBook. In Chapter 1, I'll give you a quick overview of the C language and provide some guidance on how to make good use of this course.

What is C?

C is a general-purpose compiled programming language. It was first developed by Dennis Ritchie in the late '60s and early '70s. The C language is widely used for all kinds of programming: everything from general-purpose applications, programming language tools and compilers – even operating systems. The C language is also widely used for programming hardware devices.

A C compiler (and an associated tool called a 'linker') is the program that translates your source code (the text you write in an editor) into machine code that is capable of being run by your operating system. C compilers are available for all major operating systems including Windows, OS X and Linux.

Editors and IDEs

In order to write C code you will need a programming editor or IDE (Integrated Development Environment) and a C compiler. You will find links to some editors and compilers (many of which are free) in the appendix of this eBook. In this course, I will generally use the CodeLite editor which is freely available for several operating systems: http://codelite.org/

Get the source code

The source code of the projects described in this course is provided in the form of a downloadable Zip archive (available on the course Home Page). You will need to unzip the archive using an UnZip tool before using the code.

MAKING SENSE OF THE TEXT

In **The Little Book Of C**, any C source code is written like this:

```
int add( int num1, int num2 ) {
    num1 = num1 + num2;
    return num1;
}
```

Any output that you may expect to see on screen when a program is run is shown like this:

```
The result of that calculation is 24!
```

When there is a sample program to accompany the code, the program name is shown in a little box like this:

01_HelloWorld

When an important name or concept is introduced, it may be highlighted in the left margin like this:

FUNCTIONS

Explanatory notes (which generally provide some hints or give a more in-depth explanation of some point mentioned in the text) are shown in a shaded box like this:

> This is an explanatory note. You can skip it if you like – but if you do so, you may miss something of interest…!

About the Author

Huw Collingbourne has been a programmer for more than 30 years. He is a top-selling online programming instructor with successful courses on C#, Object Pascal, Ruby, JavaScript and other topics. For a full list of available courses be sure to visit the Bitwise Courses web site: http://bitwisecourses.com/

He is author of *The Book Of Ruby* from No Starch Press and he holds the position of Director of Technology at SapphireSteel Software, makers of the *Ruby In Steel*, *Sapphire* and *Amethyst* programming environments for Microsoft Visual Studio (http://www.sapphiresteel.com/).

He is a well-known technology writer in the UK and has written numerous opinion and programming columns (including tutorials on C#, C++, Delphi, Java, Smalltalk and Ruby) for a number of computer magazines, such as *Computer Shopper*, *Flash & Flex Developer's Magazine*, *PC Pro*, and *PC Plus*. He is author of the free eBook *The Little Book of Ruby* and is the editor of the online computing magazine Bitwise (http://www.bitwisemag.com/).

In the 1980s he was a pop music journalist and interviewed most of the New Romantic stars, such as Duran Duran, Spandau Ballet, Adam Ant, Boy George, and Depeche Mode. He is now writing a series of New Romantic murder mystery novels.

At various times Huw has been a magazine publisher, editor, and TV broadcaster. He has an MA in English from the University of Cambridge and holds a 2nd dan black belt in aikido, a martial art which he teaches in North Devon, UK (http://hartlandaikido.co.uk/). The aikido comes in useful when trying to keep his Pyrenean Mountain Dogs under some semblance of control.

Chapter 2 – First Programs

Once you have a C compiler and a C source code editor installed you are ready to start programming in C. That's what we'll do in this chapter.

Hello World

This is the traditional "Hello World" program in C…

01_HelloWorld

```
#include <stdio.h>

main() {
    printf("hello world\n");
}
```

This program uses (that is, it 'includes') code from the C-language 'standard input/output library, `stdio`, using this statement:

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

The code that starts with the name `main` is the 'main function' – in other words, it is the first bit of code that runs when the program runs. The function name is followed by a pair of parentheses. The code to be run is enclosed between a pair of curly brackets:

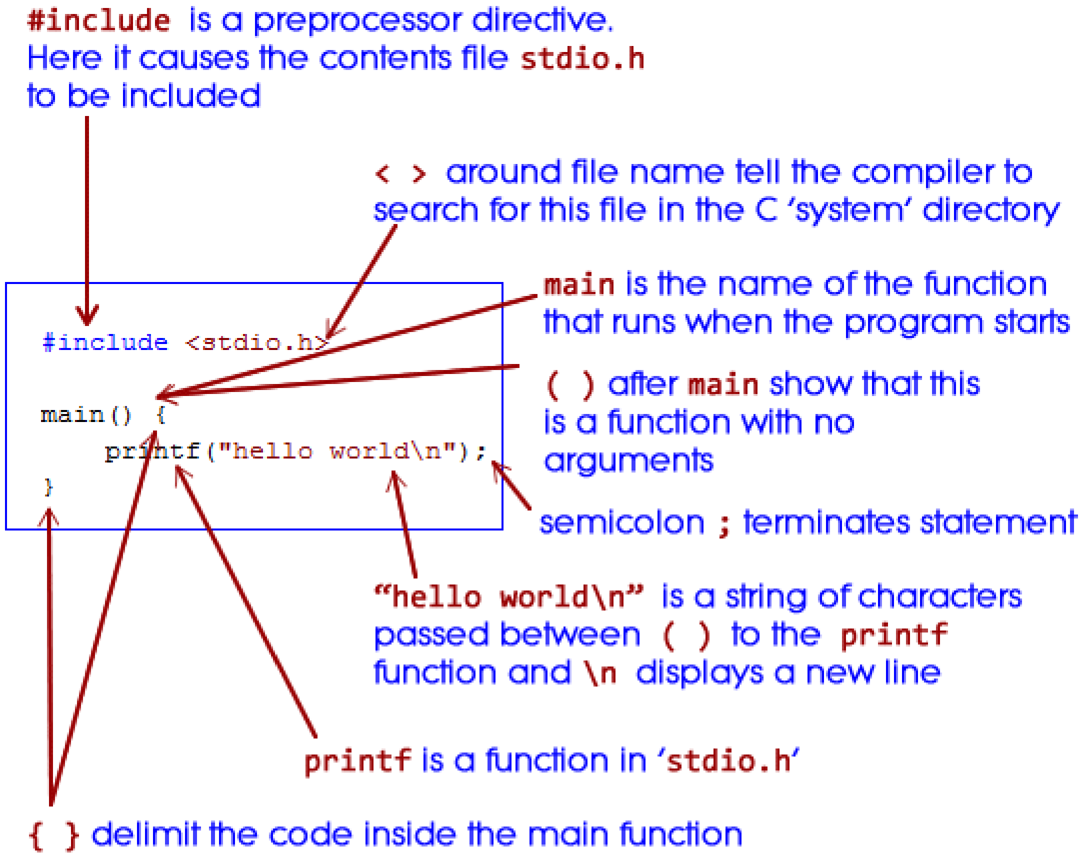
```
main() {

}
```

In this case, the code calls the C `printf` function to print the string (the piece of text) between double-quotes. The "\n" at the end of the string causes a newline to be displayed:

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

THE ANATOMY OF A C PROGRAM

This shows the essential features of the simple ‘Hello world’ program…



```
#include <stdio.h>

main() {
    printf("hello world\n");
}
```

#include is a preprocessor directive. Here it causes the contents file stdio.h to be included

< > around file name tell the compiler to search for this file in the C ‘system’ directory

main is the name of the function that runs when the program starts

( ) after main show that this is a function with no arguments

semicolon ; terminates statement

“hello world\n” is a string of characters passed between ( ) to the printf function and \n displays a new line

printf is a function in ‘stdio.h’

{ } delimit the code inside the main function

The program above could be rewritten like this:

02_HelloWorldAgain

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("hello world\n");
    return 0;
}
```

In fact, if you create a new C project using the CodeLite environment, the code above will be generated automatically. When this program is run, you will see no difference from the last program – it too displays “Hello world” followed by a newline. The main differences are that this time the name of the `main` function is preceded by `int`. This shows that the function returns an integer (a full number) when it finishes running. The number 0 is returned in the last line of the function:

```
return 0;
```

This return value is unlikely to be of any significance in your programs and, for the time being at any rate, you can ignore it. By tradition, a value of 0 just means that the program ran without any errors. Any other value might indicate an ‘error code’.

The other difference is that this program contains two ‘arguments’, called `argc` and `argv`, between parentheses:

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

These arguments may optionally be initialized with values passed to the program when it is run. I’ll shown an example of this in the next sample program.

03_HelloWorldArgs

To pass values to the program, you can just run the program at the command prompt and put any arguments (bits of data – numbers or words) after the name of the program itself, with spaces between each item. For example, if I wanted to pass the arguments “hello” and “world” to the program **03_HelloWorldArgs.exe** (on Windows) or **03_HelloWorldArgs.app** (on OS X) I would enter this at the command prompt or Terminal:

```
03_HelloWorldArgs hello world
```

My program ‘receives’ those two bits of data and it stores them in the second argument, `argv`. The first argument, `argc` is an automatically calculated value that represents the total number of the arguments stored in `argv`. This is the program code:

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) {
      printf("Hello World! argc=%d arg %d is %s\n", argc, i, argv[i]);
    }
    return 0;
}
```

When I pass the program the two arguments: `hello` and `world`, this is the output which is displayed:

```
Hello World! argc=3 arg 0 is 03_HelloWorldArgs
Hello World! argc=3 arg 1 is hello
Hello World! argc=3 arg 2 is world
```

This shows that the count (argc) of arguments is 3 even though I have only passed two arguments. That's because the program name itself, HelloWorldArgs, is automatically passed as the first argument. The first argument here has the index number 0. The arguments at index 1 and 2 are the arguments that I passed to the program: hello and world.

NOTE: the two asterisks before argv are important:

```
char **argv
```

They indicate that argv is a list of strings. Strictly speaking argv is an 'argument vector' or a pointer to an array of character-string arguments.

The block of code that starts with the keyword for is a loop that causes the code that follows it, between the curly braces, to execute for a certain number of times. Here the code executes for the number of times indicated by the value of the argc argument). The printf statement prints the string "Hello World! argc=%d arg %d is %s\n" and it substitutes the values of argc, i, argv[i], at the points marked by %d, %d and %s in the string. At each turn through the for loop the string at the index i in the argv array is printed.

PUTS AND PRINTF

There are several functions that can be used to display (print) information when your C programs run. Both printf and puts, can display a simple string.

04_printf

```
printf("hello world\n");
puts("hello world again\n");
```

The printf function also allows you to embed 'format specifiers' into a string. A format specifier begins with a % and is followed by a letter: %s specifies a string. %d specifies a decimal or integer. When format specifiers occur in the string, the string must be followed a comma-delimited list of values. These values will replace the specifiers in the string. The programmer must take care that the values in the list exactly match the types and the number of the format specifiers in the string otherwise the program may crash. Here is an example:

```
printf("There are %d bottles standing on the %s.\n", 20, "wall\n" );
```

When run, the code produces the following output:

```
There are 20 bottles standing on the wall
```

Comments

It is a good idea to add comments to your programs to describe what each section is supposed to do. C lets you insert multi-line comments between pairs of `/*` and `*/` delimiters, like this:

03_HelloWorldArgs

```
/* This program displays any
 * arguments that were passed to it */
```

In addition to these multi-line comments, modern C compilers also let you use ‘line comments’ that begin with two slash characters `//` and extend to the end of the current line. Line comments may either comment out an entire line or any part of a line which may include code before the `//` characters. These are examples of line comments:

```
// This is a full-line comment
for (i = 0; i < argc; i++)  // this comment follows some code
```

Chapter 3 – Fundamentals of C

When you want to store values in your programs you need to declare variables. A variable is simply a name (more formally, we'll call it an 'identifier') to which some value can be assigned. A variable is like the programming equivalent of a labelled box. You might have a box labelled '*Petty Cash*' or a variable named `pettycash`. Just as the contents of the box might vary (as money is put into it and taken out again), so the contents of a variable might change as new values are assigned to it. You assign a value using the equals sign (`=`).

Variables and Types

In C a variable is declared by stating its data-type (such as `int` for an integer variable or `double` for a floating-point variable) followed by the variable name. You can invent names for your variables and, as a general rule, it is best to make those names descriptive.

This is how to declare a floating-point variable named `mydouble` with the `double` data-type:

```
double mydouble;
```

You can now assign a floating-point value to that variable:

```
mydouble = 100.75;
```

Alternatively, you can assign a value at the same time you declare the variable:

```
double mydouble = 100.75;
```

Floating-point numbers

There are several data types which can be used when declaring floating point variables in C. The `float` type represents single-precision numbers; `double` represents double-precision numbers and `long double` represents higher precision numbers. In this course, I shall normally use `double` for floating-point variables.

Integers and Floats

Now let's look at a program that uses integer and floating point variables to do a calculation. My intention is to calculate the grand total of an item by starting with its subtotal (minus tax) and then calculating the amount of tax due on it by multiplying that subtotal by the current tax rate. Here I'm assuming that tax rate to be 17.5% or, expressed as a floating point number, 0.175. Then I calculate the final price – the grand total – by adding the tax onto the subtotal. This is my program:

01_Calc

```
#include <stdio.h>

int main(int argc, char **argv) {
    int subtotal;
    int tax;
    int grandtotal;
    double taxrate;

    taxrate = 0.175;
    subtotal = 200;
    tax = subtotal * taxrate;
    grandtotal = subtotal + tax;

    printf( "The tax on %d is %d, so the grand total is %d.\n",
                                  subtotal, tax, grandtotal );
    return 0;
}
```

Once again, I use `printf` to display the results. Remember that the three place--markers, `%d`, are replaced by the values of the three matching variables: `subtotal`, `tax` and `grandtotal`.

When you run the program, this is what you will see:

```
The tax on 200 is 34, so the grand total is 234.
```

But there is a problem here. If you can't see what it is, try doing the same calculation using a calculator. If you calculate the tax, *200 * 0.175*, the result you get should be *35*. But my program shows the result to be *34*.

This is due to the fact that I have calculated using a floating-point number (the `double` variable, `taxrate`) but I have assigned the result to an integer number (the `int` variable, `tax`). An integer variable can only represent numbers with no fractional part so any values after the floating point are ignored. That has introduced an error into the code.

The error is easy to fix. I just need to use floating-point variables instead of integer variables. Here is my rewritten code:

02_Calc

```
#include <stdio.h>

int main(int argc, char **argv) {
    double subtotal;
    double tax;
    double grandtotal;
    double taxrate;

    taxrate = 0.175;
    subtotal = 200;
    tax = subtotal * taxrate;
    grandtotal = subtotal + tax;

    printf( "The tax on %.2f is %.2f, so the grand total is %.2f.\n",
                                    subtotal, tax, grandtotal );
    return 0;
}
```

This time all the variables are doubles so none of the values is truncated. I have also used the float `%f` specifiers to display the float values in the string which I have passed to the `printf` function. In fact, you will see that the format specifiers in the string also include a dot and a number numbers like this: `%.2f`. This tells `printf` to display at least two digits to the right of the decimal point.

You can also format a number by specifying its width – that is, the minimum number of characters it should occupy in the string. So if I were to write `%3.2` that would tell `printf` to format the number in a space that takes up at least 3 characters with at least two digits to the right of the decimal point. Try entering different numbers in the format specifiers (e.g. `%10.4f`) to see the effects these numbers have. Here are examples of numeric formatting specifiers that can be used with `printf`:

NUMERIC FORMAT SPECIFIERS

`%d`	print as decimal integer
`%4d`	print as decimal integer, at least 4 characters wide
`%f`	print as floating point
`%4f`	print as floating point, at least 4 characters wide
`%.2f`	print as floating point, 2 characters after decimal point
`%4.2f`	print as floating point, at least 4 wide and 2 after decimal point

CONSTANTS

If you want to make sure that a value cannot be changed, you should declare a constant. A constant is an identifier to which a value is assigned but whose value (unlike the value of a variable) should never change during the execution of your program. The traditional way of defining a constant is C is to use the preprocessor directive `#define` followed by an identifier and a value to be substituted for that identifier. Here, for example, is how I might define a constant named `PI` with the value 3.141593

```
#define PI 3.141593
```

In fact, the value of a constant defined in this way is not absolutely guaranteed to be immune from being changed by having a different value associated with the identifier. In C, the following code is legal (though your compiler may show a warning message):

```
#define PI 3.14159
#define PI 55.5
```

Modern C compilers provide an alternative way of defining constants using the keyword `const`, like this:

```
    const double PI = 3.14159;
```

If I try to assign a new value to this type of constant the compiler won't let me. It shows an error message, so this is *not* permitted:

```
          const double PI = 3.14159;
          const double PI = 55.5;
```

In the sample program, **03_Calc**, I have shown three alternative versions of my tax calculator using, first, the variable `taxrate` to store the tax rate to be used in calculations, then the `#define` constant `TAXRATE_DEFINED`:

03_Calc

```
#define TAXRATE_DEFINED 0.175
```

And, finally, the const constant, `TAXRATE_CONST`:

```
const double TAXRATE_CONST = 0.175;
```

Probably most C programmers would use the `#define` version (as this has become the 'traditional' way of defining C constants). But in fact only the value assigned to the constant defined as a `const` is completely protected from being altered subsequently.

Naming conventions

You may have noticed that I have named my constants (both those declared using `const` and those declared using `#define`) in capital letters like this: `TAXRATE_CONST`, `TAXRATE_DEFINED`, `PI`.

Naming constants in all capital letters with the individual 'words' separated by underscores is a widely adopted convention in C. Bear in mind, however, that it is only a convention and not a rule.

When you write C code you may want to adopt some other conventions when writing the names of variables and functions. These are some common conventions:

Use lowercase for variable names:

e.g.

```
  int tax;
```

Use lowercase for function names:

e.g.

```
  int calculate() {
  }
```

Use lowercase for parameter names (items between parentheses that are passed to a function):

e.g.

```
  int calculate_grand_total( int subtotal )
```

Use underscores to separate 'words' in function and variable names:

e.g.

```
  int calculate_grand_total( int subtotal ) {
      int grand_total;
      grand_total = subtotal + SERVICE_CHARGE;
      return grand_total;
  }
```

Once again, let me emphasise that these are only conventions. You are not obliged to name your functions and variables in this way. In fact, some programmers adopt other conventions. For example, some people prefer to used mixed-case letters to divide 'words' instead of underscores, like this:

```
int calculateGrandTotal( int subTotal ) {
```

Whichever naming convention you choose, try to be consistent. In this course, I will generally use lowercase letters separated by underscores for the names of functions, uppercase letters for constants and mixed-case letters for parameters and variables. But that is just my choice and other programmers will adopt other naming conventions.

> IMPORTANT: Be aware that C is a case-sensitive language, so a variable called `subtotal` is treated as a different variable from one called `subTotal` or one called `subTOTAL`. Similarly, a function that is named `my_function` is different from one named `my_Function` or `My_Function`.

Another important point when naming variables and functions is to choose names that describe what those variables and functions actually do. Look at this code:

```
#define v 33

int myfunc( int z ) {
    int t;
    t = z + v;
    return t;
}
```

It tells you nothing at all about what the function is intended to do. Below I have rewritten this function. The two versions (the one above and the one below) do exactly the same operations. However, I think you will agree that the one shown below is much easier to make sense of!

```
#define SERVICE_CHARGE 33

int calculate_grand_total( int subtotal ) {
    int grand_total;
    grand_total = subtotal + SERVICE_CHARGE;
    return grand_total;
}
```

Chapter 4 – Conditional tests, Operators and Input

In your programs you will often want to assign values to variables and, later on, test those values. For example, you might write a program in which you test the age of an employee in order to calculate his or her bonus. Here I use the 'greater than' operator > to test of the value of the age variable is greater than 45:

```
if (age > 45) {
      bonus = 1000;
}
```

We've already used several operations such as the addition operator +, the multiplication operator * and the assignment operator = in code like this (from Chapter 3):

```
tax = subtotal * TAXRATE_DEFINED;
grandtotal = subtotal + tax;
```

The time has come to look at C's operators in a bit more detail.

Operators

Operators are special symbols that are used to do specific operations such as the addition and multiplication of numbers. One of the most important operators is the assignment operator, =, which assigns the value on its right to a variable on its left. Note that the type of data assigned must be compatible with the type of the variable.

This is an assignment of an integer (10) to an int variable named myintvariable:

```
int myintvariable;
myintvariable = 10;
```

ASSIGNMENT OR EQUALITY?

Beware. While *one* equals sign `=` is used to assign a value, *two* equals signs `==` are used to test a condition.

`=` this is the assignment operator.

e.g. `x = 1;`

`==` this is the equality operator.

e.g. `if (x == 1)`

TESTS AND COMPARISONS

C can perform tests using the `if` statement. The test itself must be contained within parentheses and it should be capable of evaluating to true or false. If true, the statement following the test executes. Optionally, an `else` clause may follow the `if` clause and this will execute if the test evaluates to false. Here is an example:

00_Operators

```
if (age > 45) {
    bonus = 1000;
} else {
    bonus = 500;
}
```

You may use other operators to perform other tests. For example, this code tests if the value of `age` is less than or equal 70. If it is, then the conditional evaluates to **true** and "You are one of our youngest employees!" is displayed. Otherwise the condition evaluates to **false** and nothing is displayed:

```
if (age <= 70){
    printf("You are one of our youngest employees!\n");
}
```

Notice that the `<=` operator means ‘less than or equal to’. It performs a different test than the `<` operator which means ‘less than’. In the sample project, the value of `age` is 70. Edit the test to use the `<` operator like this and run the program again to see the difference:

```
if (age <= 70)
```

These are the most common comparison operators that you will use in tests:

```
==        // equals
!=        // not equals
>         // greater than
<         // less than
<=        // less than or equal to
>=        // greater than or equal to
```

Compound Assignment Operators

Some assignment operators in C perform a calculation prior to assigning the result to a variable. This table shows some examples of common 'compound assignment operators' along with the non-compound equivalent.

operator	example	equivalent to
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`

It is up to you which syntax you prefer to use in your own code. Many C and C++ programmers prefer the terser form as in: `a += b`. But the same effect is achieved using the slightly longer form as in: `a = a + b`.

Increment ++ and Decrement -- operators

When you want to increment or decrement by 1 (add 1 to, or subtract 1 from) the value of a variable, you may also use the `++` and `--` operators. Here is an example of the increment (`++`) operator:

```
int a;
a = 10;
a++;      // a is now 11
```

This is an example of the decrement (`--`) operator:

```
int a;
a = 10;
a--;      // a is now 9
```

Prefix and Postfix Operators

You may place these operators either before or after a variable like this: `a++` or `++a`. When placed before a variable, the value is incremented before any assignment is made:

```
num1 = 10;
num2 = ++num1;              // num2 = 11, num1 = 11
```

When placed after a variable, the assignment of the existing value is done before the variable's value is incremented:

```
num1 = 10;
num2 = num1++;              // num2 = 10, num1 = 11
```

As a general rule, I would recommend that you stick to using `++` and `--` as postfix operators. In fact, there is often nothing wrong with using the longer form `a = a + 1` or `a = a - 1`. Mixing prefix and postfix operators in your code can be confusing and may lead to hard-to-find bugs. So, whenever possible, keep it simple and keep it clear.

IF..ELSE

At the start of the chapter, I gave a simple example of testing a condition using `if`, like this:

```
if (age > 45)
```

In that example, the value of `age` (70) was assigned in my code and so it was bound to be the same value every time the program was run. In a real-world program, that would is not very useful. In **01_TestAge** project, I have created a simple program that prompts the user to enter their age so the value of `age` may be different each time the program runs:

01_TestAge

```
int main(int argc, char **argv) {
    char agestring[10];
    int age;
    int bonus;

    printf("Enter your age : ");
    gets(agestring);
    age = atoi(agestring);
    if (age > 45) {
          bonus = 1000;
    } else {
          bonus = 500;
    }
    printf("Your age is %d, so your bonus is %d.\n", age, bonus);

    return(0);
}
```

In order to get input from the command prompt I have used the `gets()` function. This gets the text that was entered at the prompt. It assigns this to the variable between parentheses. This variable is treated as a string (a sequence of alphanumeric characters). In this case, my variable, `agestring`, has been declared as an array of 10 characters:

```
char agestring[10];
```

But my `if` test requires an integer variable. I convert the string `agestring` to an integer using the `atoi()` function and I assign the result to the `int` variable named `age`:

```
age = atoi(agestring);
```

Once that is done, the value of `age` can be tested against an integer value:

```
if (age > 45) {
    bonus = 1000;
} else {
    bonus = 500;
}
```

But there is a problem here. What if the user enters some text that cannot be converted to an integer? Say, for example, when prompted to enter an age, the user enters “Forty”? The `atoi()` function can only convert a string that contains integer characters such as “40”. When it tries to convert anything else it fails and it returns the value 0. In my program, a return value of 0 is treated as a valid age, a bonus will be applied and this message will be displayed:

```
Your age is 0, so your bonus is 500.
```

That is clearly not what I intended. So how do I fix this problem?

Note: atoi will work with values that commence with a number and end with a string such as “123xyz”. It silently ignores the “xyz” when converting the numeric part of the string. atoi is an old function which I use here because it is provided by a very large number of compilers. Modern C compilers also provide alternative conversion functions such as strtol, atoi_1 and others. Search your compiler’s documentation to see which conversion routines it recommends.

02_TestAge

One simple solution to the problem is to test if the value of is age 0 and take specific action if it is (for example, display an error message) and only if it is not 0 continue to perform the regular test. I have done this in **02_TestAge**. This illustrates how to ‘nest’ if and else tests:

```
if (age == 0) {       // if #1
    printf("You entered an invalid age, so your bonus cannot be
                calculated.\n");
} else {              // else #1
    if (age > 45) {         // if #2
          bonus = 1000;
    } else {                // else #2
          bonus = 500;
    }
    printf("Your age is %ds, so your bonus is %d.\n", age, bonus);
}
```

Here, I test if the value of age is 0. If it is this message is displayed: *"You entered an invalid age, so your bonus cannot be calculated.\n"*. If age has any other value the code block following the first else executes. This block contains all the code between the outermost set of curly brackets:

```
else {
    if (age > 45) {
          bonus = 1000;
    } else {
          bonus = 500;
    }
    printf("Your age is %ds, so your bonus is %d.\n", age, bonus);
}
```

This code includes another if..else test. Remember that this second if..else test only executes when the first if test fails (if age was not 0). These types of ‘nested’ if..else tests are not uncommon in C programs. However, don’t overdo them. If you have `if..else` tests that are 'nested' more than one-level deep, as in the code above, the logic of the tests may be quite hard to understand. If you find you've written code with many nested levels of `if` or `if..else` tests you should consider rewriting the code to make it easier to understand.

GETS() AND FGETS()

The last program may seem to work as expected. However, there is a potential problem. It turns out that the `gets()` function will go right ahead and assign any amount of text that the user happens to enter at the command prompt to the variable specified. Recall that I have defined the `agestring` variable to have a maximum of 10 characters. What happens, then, if the user enters 20 or 25 characters? To clarify this problem try out the **03_GetInput** program.

> In this section I explain a number of problems I have found and handled when developing line-reading functions. Developers often write code in this way – by rewriting and refining their code to deal with problems as they arise. If you are a novice programmer you may find some of this section hard to understand at first reading. Don't worry. I will explain much of the technicality of this code – such as functions, arguments and null-terminated strings - in later chapters.

Look at the function named `getinput_with_gets()`:

03_GetInput

```
void getinput_with_gets() {
    char firstname[5];
    char lastname[5];
    printf("Enter your first name:");
    gets(firstname);
    printf("Enter your last name:");
    gets(lastname);
    printf("Hello, %s, %s\n", firstname, lastname);
}
```

Here `gets()` tries to read in all the characters that have been entered but it only assigns a fixed number (5) of those characters to the fixed-length array, `firstname`. That's fine if I only enter 4 characters (the 5th character will be the carriage return). But if I enter more than 4 characters then I will have a problem. That's because the data that I enter spills over the end of the memory used by the 5-character array that I has been declared to hold that data – and the overspilled characters will end up in some unpredictable area of my computer's memory. The chances are that those characters will overwrite some bit of memory that is already being used for something else. At best, this means that the data which I assign to my variables may be incorrect (it may include some of the 'overspill characters'). At worst, it could mean that my program crashes. For that reason, even though the `gets()` function may seem easy to use, I would not encourage you to do so.

The alternative `fgets()` function is a safer alternative to `gets()`. The `fgets()` function takes three 'arguments' between parentheses: 1) the array to which the data will be assigned, 2) the maximum number of characters to be read and 3) the name of the data-source or 'stream' from which to read them. The data-source may be a file on disk. Alternatively, the name `stdin` indicates that the source is the 'standard input' – here, that means any text entered by the user. The value specified for the maximum number of characters causes a string to be truncated at the specified number –1. Let's assume you enter value 5 as shown below:

```
fgets(firstname, 5, stdin);
```

NULL-TERMINATED STRINGS

If you now enter 5 characters at the command prompt like this: *'abcde'*, only the first 4 characters, *'abcd'*, will be read. That is because the 5th character is a special null character (`'\0'`) that is automatically appended to a string. A null character marks the end of a string. This will be explained in Chapter 7.

There is still one problem, however. If you enter more characters than are actually processed by your code, those characters remain in memory, waiting to be processed. So when you next try to read some characters from the command line, the characters waiting to be processed will be read in first. Let's look at an example. Assume you have this code:

```
char firstname[5];
char lastname[5];
printf("Enter your first name:");
fgets(firstname, 5, stdin);
printf("Enter your last name:");
fgets(lastname, 5, stdin);
printf("Hello, %s, %s\n", firstname, lastname);
```

When you enter *'abcde'* and press the Enter key, the first 4 characters, *'abcd'*, are assigned to the variable `firstname`. You probably now expect that you will be prompted to enter your last name. But that is not what happens. What happens is that the two unprocessed characters (the `'e'` plus the carriage return `'\n'` that you entered previously) are immediately processed by the next `fgets()` and those two characters are assigned to the `lastname` variable. The final `printf()` then displays the following:

```
Hello, abcd, e
```

Flushing Buffers

In order to fix this we need to 'use up' the extra characters that were entered but were not assigned to the `firstname` variable. To use programming jargon, the spare characters are said to be stored in an array of characters called a 'buffer' and in order to use them up we need to empty of 'flush' that buffer. The simplest way to do that is to read through the remaining characters in the buffer. In this case, I want to read all the characters up until a newline `'\n'` (which indicates the end of input entered at the command prompt). In fact, I also test for the end of a file (a negative integer value assigned to the `EOF` constant) which ensures compatibility with attempts to read lines from a disk file. This is my buffer-flushing function:

```
void flush_input(){
    int ch;
    while ((ch = getchar()) != '\n' && ch != EOF);
}
```

The `!=` operator here means 'not equal to' so the `while` loop continues reading characters while the character that is read is not a newline character `('\n')` and (`&&`) it is also not at the end of a file (`EOF`).

Note: The standard C library function `fflush()` also flushes a buffer. However, it is defined to flush the output buffer (the characters being written) rather than the input buffer (the characters being read). Some implementations of `fflush()` work with both input and output but this cannot be relied upon, which is why it may be safer to write your own input flushing routine.

But there remains yet another problem with my code. The `fgets()` function reads in all the text entered including the newline `'\n'` character. That is not a problem if I enter more characters than will fit into my fixed-length arrays, `firstname` and `lastname`. In that case, the variables are initialized with the appropriate number of characters and a null terminator is automatically appended by `fgets()`. But if I enter fewer characters than are needed to initialize the variables, `fgets()` reads the newline character which was entered when I pressed the Enter key. Then I call my `flush_input()` function. This carries on reading characters *until a newline is found*. But there is no longer a newline to be read because I have already processed that character. As a consequence I need to press the Enter key a second time. I could rewrite the code to try to deal with this but I'm already starting to find it confusing. And when your code starts to be confusing, it's time to simplify it!

I've decided I need a line-reading function that handles all the complexity. I want it to read a line of text up to a specified maximum length. I also want it to work in exactly the same way whether I enter a short bit of text, a long bit of text or even no text at all (should I just press the Enter key). As an added extra it might be useful if it also returned the length of the text read, just in case I need to verify the length for some reason. That is what I have done in **04_Readln**.

Below is my first effort at writing a function which safely reads a line of text and assigns it to an array of characters (declared in the function header as `char s[]`).

04_Readln

```
int readln (char s[], int maxlen) {
    int len_s;
    fgets(s, maxlen, stdin);
    len_s = strlen(s);
    if (s[len_s - 1] == '\n') {
          s[len_s - 1] = '\0';
          len_s -= 1;
    }
    rewind(stdin); // This flushes the keyboard buffer
    return len_s;
}
```

This function once again uses `fgets()` to get a string. The `maxlen` parameter specifies the maximum number of characters to read. I use the built-in function `strlen()` to get the length of the string. The `if` block tests if the character at the end of the string (that is, at the position given by `len_s -1`) is a newline character `(‘\n’)` and, if so, it replaces this will a null character `(‘\0’)`. Remember that the null character is used in C to indicate the end of a string.

Finally I have used the `rewind()` function to flush the keyboard buffer instead of my own `flush_input()` function. It turns out that `rewind()` is used to move the focus back to the beginning of a file (technically, it repositions the file pointer to the beginning of a file). However, when `stdin` is passed to the function, `rewind()` has the effect of clearing the keyboard buffer. The only problem here is that the `rewind()` function is not provided on all operating systems. It works fine on Windows. On some operating systems a function such as `tcflush()`may be used instead.

Alternatively, you may prefer to write a routine that does not rely upon any of these non-standard functions. That is what I have done in this alternative version of `readln()`:

```
   int readln(char s[], int maxlen) {
   char ch;
   int i;
   int chars_remain;
   i = 0;
   chars_remain = 1;
   while (chars_remain) {
         ch = getchar();
         if ((ch == '\n') || (ch == EOF) ) {
               chars_remain = 0;
         } else if (i < maxlen - 1) {
               s[i] = ch;
               i++;
         }
   }
   s[i] = '\0';
   return i;
}
```

This reads the text entered at the keyboard one character at a time. All the reading is done using `getchar()` so I no longer have to worry about any side-effects due to newline characters that may, or may not, have been handled by the `fgets()` function. Since my function now processes every character it can both assign the characters required to initialize variables and also process any extra characters in order to flush the keyboard buffer.

This is how it works. The `while` loop continues getting characters as long as any characters remain to be processed. I use an integer variable, `chars_remain`, to test whether the loop condition is *true* or *false*. This variable has the default value 1 (in C a non-zero value is regarded as meaning *true* when it is tested) and it becomes 0 (equivalent to *false*) when there are no more characters left. Here it becomes *false* when the newline (or `EOF`) are found. A newline will be found at the end of the input string – at the point at which I pressed the Enter key. Until the newline is found the while loop continues running.

At each turn through the loop the `i` counter variable is incremented. The code fills the character array `s[]` by appending each character that has been read as long as `i` is less than `maxlen -1` (here 1 is subtracted to allow for the terminating null character). The value of `maxlen` indicates the length of the character array that was passed to the `readln()` function. Once `i` is equal to or greater than `maxlen`, the code stops adding characters to `s[]`. However, that doesn't mean that the `while` loop stops running. Recall that this loop only stops running when a newline or `EOF` are found – and that might be either *before* or *after* I have finished adding characters to `s[]`. If it is after `s[]` has been fully initialized then the reading of characters in the loop just serves the purpose of flushing the buffer – that is, reading and ‘using up’ any characters that I don’t need.

Finally, I put a null character `'\0'` at the end of the `s[]` array. That marks the end of the string. And I return the value of `i` which gives the length of the string – not counting the terminating null character.

Logical Operators

In some of the code examples in this chapter, I have used the `&&` operator to mean ‘and’ and the `||` operator to mean ‘or’. The `&&` and `||` operators are called ‘logical operators’. Logical operators can help you chain together conditions when you want to take some action only when *all* of a set of conditions are true or when *any one* of a set of conditions is true. For example, you might want to offer a discount to customers only when they have bought goods worth more than 100 dollars *and* they have also bought the deal of the day. In code these conditions could be evaluated using the logical *and* (`&&`) operator, like this:

```
if ((valueOfPurchases > 100) && (boughtDealOfTheDay))
```

But if you are feeling more generous, you might want to offer a discount *either* if a customer has bought goods worth more than 100 dollars *or* has bought the deal of the day. In code these conditions can be evaluated using the logical *or* (`||`) operator, like this:

```
if ((valueOfPurchases > 100) || (boughtDealOfTheDay))
```

Boolean values

Logical operators test Boolean values. A Boolean value can either be *true* or it can be *false*. Some programming languages have a special Boolean data type. C does not. As mentioned earlier, in C, any non-zero value such as 1 or 100 is evaluated as *true*. A zero value is evaluated as false.

It is possible to create quite complex conditions by chaining together tests with multiple `&&` and `||` operators. Be careful, however. Complex tests are often hard to understand and if you make a mistake they may produce unwanted side effects.

In addition, just as when you are using arithmetic operators, you may avoid ambiguity by grouping the individual 'test conditions' between parentheses.

The *05_LogicalOperators* sample program provides a few more examples of using logical operators. Note that the `!` operator can be used to negate a condition. So the test `!( a == b)` ('not a equals b') is equivalent to `(a != b)` ('a is not equal to b'). Similarly this test…

05_LogicalOperators

```
(number_of_children != 0)
```

…could be rewritten like this:

```
!(number_of_children == 0)
```

Chapter 5 – Functions

Functions provide ways of dividing your code into named 'chunks'. We've used functions many times throughout this course. In fact the block of code named `main()` that begins every program is a function. We've written functions with other names such as the `readln()` function from the last chapter. And we've used functions such as `printf()` which are supplied, as standard, with the C compiler. Here I want to explain functions in more detail.

Function Declarations

A function is declared by specifying the data type of any value that's returned by the function (such as `int` or `char`) or `void` if nothing is returned. Then comes the name of the function, which may be chosen by the programmer. Then comes a pair of parentheses.

The parentheses may contain one or more 'arguments' or 'parameters' separated by commas. The argument names are chosen by the programmer and each argument must be preceded by its data type. When a function returns some value to the code that called it, that 'return value' is indicated by preceding it with the `return` keyword.

Here are some example functions:

01_Functions

A function that takes no arguments and returns nothing:

```
void sayHello( ) {
    printf( "Hello\n" );
}
```

A function that takes a single string (an array of `char`) argument and returns nothing:

```
void greet( char aName[] ) {
    printf( "Hello %s\n", aName );
}
```

A function that takes two `int` arguments and returns an `int`:

```
int add( int num1, int num2 ) {
    num1 = num1 + num2;
    return num1;
}
```

A function that takes two `int` arguments and returns a `double`:

```
double divide( int num1, int num2 ) {
    return num1 / num2;
}
```

Note: In other languages, functions that return nothing (`void`) may be called ‘subroutines’ or ‘procedures’. In Object Oriented languages such as C#, Java and C++, functions that are ‘bound into’ objects are called ‘methods’. C is not object-oriented so the term ‘method’ is not used. In fact, C programmers generally refer to all functions – both those that return values and those that do not - simply as ‘functions’.

To execute the code in a function, your code must ‘call’ it by name. In C, to call a function with no arguments, you must enter the function name followed by an empty pair of parentheses and a semicolon like this:

```
sayHello( );
```

To call a function with arguments, you must enter the function name followed by the correct number and data-type of values or variables (separated by commas), like this:

```
greet("Fred");
divide( 100, 3 );
```

If a function returns a value, that value may be assigned (in the calling code) to a variable of the matching data type. Here, `add()` function returns an `int` and this is assigned to the `int` variable called `total`:

```
total = add( n1, n2 );
```

The `divide()` function returns a `double` and this is assigned to the `double` variable called `result`:

```
result = divide( 100, 3 );
```

ARGUMENTS ARE PASSED 'BY VALUE'

Pay close attention to the code in this function:

01_Functions

```
int add( int num1, int num2 ) {
    num1 = num1 + num2;
    return num1;
}
```

Notice that I have assigned the value of the addition of the arguments `num1 + num2` to one of the arguments, `num1`. My code, in `main()` calls this function like this…

```
n1 = 10;
n2 = 3;
total = add( n1, n2 );
```

When the variables `n1` and `n2` are passed to the `add()` function, their values (10 and 3) are assigned to the named parameters, `num1` and `num2`. Then 10 and 3 are added together and assigned to `num1`. That means that `num1` now has the value 13. So does that mean that `n1`, the argument that matches the `num1` parameter, also has the value 13?

When you run the code you will see that it does not. The two variables, `n1` and `n2`, in the `main()` function are unaffected by changes made to the matching parameters in the `add()` function. That is because the C language passes arguments 'by value'. In other words when `n1` and `n2` are passed to the function `add()` it is only their *values* that are sent – here these are the integer values, 10 and 3. These values are 'detached' from the original variables.

If you have programmed in other languages you may know that sometimes changes made to parameters inside a function may indeed change the values of any matching variables that were passed to that function. When that happens we say that the parameters were passed 'by reference' (since they refer to the original variables) rather than 'by value' (in which case the parameters are assigned the values passed from the original variables but do not refer directly to those variables). I'll have more to say about passing 'by reference' in later chapters of this course.

SWITCH

If you need to perform any tests, it is often quicker to write them as 'switch statements' instead of multiple `if..else` tests. In C a `switch` statement begins with the keyword `switch` followed by a test value. This test value must be an integer or some piece of data (such as a `char`) that may be treated as an integer. Then you write one or more `case` statements, each of which specifies a value that is compared with the test value. If a match is made (when the `case` value and the test value are the same), the code following the `case` value and a colon (e.g. `case 10:`) executes. When you want to exit the code in a `case` block you need to use the keyword `break`. If `case` tests are not followed by `break`, sequential `case` tests will be done one after the other until `break` is encountered. You may specify a `default` which will execute if no match is made by any of the `case` tests. In the **02_SwitchCase** project, I use a `for` loop to count through values from 0 to 127 and display the ASCII character which that number represents. At each turn through the loop, I use a `switch` statement to try to categorise the type of each character to show whether, for example, it is a Space character, a Tab character or a number.

ASCII CODES AND CHARACTERS

The acronym, ASCII, stands for American Standard Code for Information Interchange. The alphabetic and numeric characters displayed on your screen may be represented internally by a specific code number in the ASCII table.

For example, ASCII code 66 equates to the upper-case 'A' character, whereas ASCII code 97 equates to a lower-case 'a'. There are also ASCII code for invisible characters such as spaces (ASCII 32) and Carriage Returns (ASCII 13). There is even a backspace character (ASCII 8) which is generated by the backspace key on your keyboard.

It is useful to know all the available ASCII values in order to be able to manipulate the characters in strings. For example, you could perform arithmetical operations using ASCII values in order to encrypt the characters in a password. There are 128 characters in the standard ASCII table numbered from 0 to 127 and there are 256 characters in the 'extended' ASCII table numbered from 0 to 255.

Here is the switch test I used to try to categorise characters with an ASCII value given by the `int` variable `i`:

02_SwitchCase

```
switch( i ) {
    case 0:
          chartype = "NULL";
          break;
    case 7:
          chartype = "Bell";
          break;
    case 8:
          chartype = "BackSpace";
          break;
    case 9:
          chartype = "Tab";
          break;
    case 10:
          chartype = "LineFeed";
          break;
    case 13:
          chartype = "Carriage Return";
          break;
    case 32:
          chartype = "Space";
          break;
    case 48:
    case 49:
    case 50:
    case 51:
    case 52:
    case 53:
    case 54:
    case 55:
    case 56:
    case 57:
          chartype = "Number";
          break;
    default:
          chartype = "Character";
          break;
}
```

In this example, **if** the `int` variable `i` is `0`, it matches the first `case` test and the string variable `chartype` will be assigned the value `"NULL"`. Then the keyword `break` is encountered so no more tests are done. Similar tests and assignments are performed when `i` has the values 7, 8, 9, 10, 13 or 32. If `i` has any value between 48 and 57, the string `"Number"` is assigned to the `chartype` variable. That is because there is no `break` after any of the `case` values from 48 to 56, so if any match is made with one of those values the execution ‘trickles down’ through all the other `case` blocks until it meets the first `break` statement – this occurs after `case 57`. If `i` has any other value – that is, any value that does not match one of the numeric case values, the code in the `default` section is run, and `"Number"` is assigned to `chartype`.

The `default` section is optional. Note that, strictly speaking you don't need to put a `break` in the `default` section. However, it is generally considered to be good practice to do so.

In C, the char data type is compatible with the `int` data type since each character is defined by an integer value. I have provided an alternative to the code above in which an `int` argument passed to the `findchartype2()` function is assigned to the char parameter `c`. My the case expressions in my `switch` block attempt to match `c` with char values (characters such as '0' between single quotation marks or non-printing characters preceeded by `\` such as the null character `'\0'`, the Tab character `'\t'` and the linefeed character `'\n'`).

```
void findchartype2( char c ) {
    switch( c ) {
          case '\0':
                chartype = "NULL";
                break;
          case '\t':
                chartype = "Tab";
                break;
          case '\n':
                chartype = "LineFeed";
                break;
          case '\r':
                chartype = "Carriage Return";
                break;
          case ' ':
                chartype = "Space";
                break;
          case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9':
                chartype = "Number";
                break;
          default:
                chartype = "Character";
                break;
    }
}
```

Ranges

Instead of writing separate `case` tests for sequential values such as '0' to '9' or 48 to 57, as in my code examples, some modern C compilers let you specify a range of values in a single `case` test. A range is defined by a start value and an end value separated by three dots. A match will be made when the `switch` test value (such as `i` or `c` in my examples) matches any values between the start and the end values of a range. For example, in the ASCII table, uppercase letters start with the value 65 ('A') and end with 90 ('Z'); lowercase letters start with 97 ('a') and end with 122 ('z'). So, when using a C compiler that supports ranges, I could write these `case` statements to test an `int` value:

```
case 65 ... 90:
   chartype = "Uppercase Letter";
   break;
case 97 ... 122:
 chartype = "Lowercase Letter";
   break;
```

Similarly, I could use these statements to test a `char` value:

```
case 'A' ... 'Z':
   chartype = "Uppercase Letter";
   break;
case 'a' ... 'z':
   chartype = "Lowercase Letter";
   break;
```

Bear in mind, however, that not all C compilers permit this syntax.

Chapter 6 – Arrays and 'while' loops

We've already used arrays of characters in several projects in this course. For example, in Chapter 4, I wrote a function called `readln()`that created an array, `s[]`, of characters that were entered at the keyboard. Each character was assigned to the `char` variable `ch` and appended to the array at the index `i` like this:

```
s[i] = ch;
```

The end result was that if the user entered the characters 'H', 'e', 'l', 'l', 'o' these would be added sequentially to the array, `s[]`.The first character would be at the 'array index' 0, the second at index 1 and so on. You can think of an array as being like a container with a fixed number of slots numbered from 0 upwards like this:

Contents:	**'H'**	**'e'**	**'l'**	**'l'**	**'o'**
Index:	0	1	2	3	4

Notice that, since the first element in an array is at index 0, the last element is at the index given by the length of the array - 1. In the array shown above, there are five characters so the array length is 5. The first element 'H' is at index 0 and the last element 'o' is at index 4, that is the position given by the array length 5 - 1.

Arrays

Arrays aren't restricted to storing characters, however. In C, an array can contain other types of data as long as each item in the array is of the same data type. This is how to declare an array named `intarray`, that can store 5 integers:

```
int intarray[5];
```

When you want to access an element at a specific index in the array you can put the index number between square brackets like this:

```
intarray[3];
```

This lets you either obtain the existing value of the element at that index or assign a new value to that element. This is how I would assign the integer value 31 to the 4th element (that is, the element at index 3) of my array:

```
intarray[3] = 31;
```

In fact, you can initialize the values of an entire array by indexing them one at a time like this:

```
intarray[0] = 1;
intarray[1] = 11;
intarray[2] = 21;
intarray[3] = 31;
intarray[4] = 41;
```

The following line of code access the value from index 3 of the array:

```
printf( "The integer at intarray[3] is: %d\n", intarray[3]);
```

Assuming the array has been initialized as shown above, this code displays:

```
The integer at intarray[3] is: 31
```

You can also initialize an array at the time it is declared. To do that you must assign a comma-delimited list of items enclosed by curly brackets like this:

```
int intarray[5] = {1,2,3,4,5};
```

In fact, when you initialize an array in this way, you don't need to specify the number of items between square brackets – the C compiler automatically creates an array capable of holding the specified number of items. Here I declare and initialize an array that contains five integers:

01_Arrays

```
int intarray[] = {1,2,3,4,5};
```

And here are arrays that contain four doubles and six chars respectively (the sixth `char` being a null 'string terminator');

```
double doublearray2[] = {2.1, 2.3, 2.4, 2.5};
char chararray[] = {'h', 'e', 'l', 'l', 'o', '\0' };
```

Since character arrays are so common in C (they are used to define strings such as "Hello world"), there is an alternative shorthand way of assigning a string to an array of chars, like this:

```
char chararray2[] = "world";
```

When a string is assigned in this way, the null terminator '\0' is appended automatically so you don't need to add it explicitly as you do when assigning chars one at a time.

One convenient way of iterating through the elements of array is provided by a for loop. Here I iterate through the 5-element intarray, counting from 0 to 4 (that is while i is less than 5: i < 5), placing the value i + 1 multiplied by 100 into each successive 'slot' of the array:

ARRAYS AND 'FOR' LOOPS

```
for (i = 0; i < 5; i++) {
    intarray[i] = ((i + 1) * 100); // note use of parentheses here!
}
```

Since i starts with the value 0, i + 1 gives 1. So the value placed at index 0 of the array is 100. The values placed at successive positions in the array are: 200, 300, 400, 500.

WHILE LOOPS

As we've seen, for loops are useful when you have a known number of items to iterate through – from 0 to 4, say. But sometimes you may not know in advance how many items you need to process. For example, you might write some code to format the lines in a text file. But each file may contain a different number of lines so you cannot assume, in advance, that there will be a specific number of lines to process.

Or you may prompt the user to enter a response such as 'y' or 'n'. You may want a certain piece of code to keep on running if the user enters 'n' but to end if the user enters 'y'. Here is a while loop that does this:

```
while( c != 'y' ) {
    printf("\nEnter y or n: ");
    c = getchar();
    getchar();
}
```

The code inside the block delimited by the curly brackets continues to run while the char variable c is not 'y'. The user is prompted to enter a character which is read from input and assigned to c. When 'y' is entered, the test while(c != 'y') fails so the code stops running. If any other character is entered the test succeeds and the code runs once again.

Incidentally, you might wonder why I have a second call to the getchar() function. This is because when the user enters 'y' or 'n' the linefeed character itself is also entered when the Return key is pressed. My second call to `getchar()` simply 'mops up' this unwanted linefeed character. If you need a more robust way of reading characters from the keyboard, refer back to the `readln()` function from Chapter 4.

> Recall that we used `while` loops in Chapter 4 when we had the problem of 'flushing' the user input by carrying on reading any characters that the user had entered *while* more characters remained to be read:
>
> ```
> void flush_input(){
>  int ch;
>  while ((ch = getchar()) != '\n' && ch != EOF);
> }
> ```

But let's assume that I decide that the default value of c should be 'y' and I initialize the variable with that value. What happens in this code?

```
char c;
c = 'y';
printf("\ngetchar() with while loop...\n");
while( c != 'y' ) {
    printf("\nEnter y or n: ");
    c = getchar();
    getchar();
}
printf("\nThat's all folks!\n");
```

In fact, what happens is that the test `c != 'y'` fails immediately because `c` is 'y'. So none of the code inside the block between `{` and `}` can ever be executed. When it runs it displays this:

```
getchar() with while loop...

That's all folks!
```

In other words, you might say that the code in a `while` block will execute 0 or more times. Here it happens to execute 0 times. If I want to make sure that the code inside the block is executed at least once you can use a different variation: a `do..while` block. This sort of block begins with the keyword `do` and ends with the `while` followed by the test condition. The test is made at the end of the block of code rather than at the start so the code must necessarily run at least once. Here is my code rewritten with a `do..while` block:

```
c = 'y';
printf("\ngetchar() with do..while loop...\n");
do {
    printf("\nEnter y or n: ");
    c = getchar();
    getchar();
}while( c != 'y' );
printf("\nThat's all folks!\n");
```

This code displays this:

```
getchar() with do..while loop...

Enter y or n:
```

Then it waits for me to enter a character. If I enter 'y' the test fails, the code block exists and this is shown:

```
That's all folks!
```

But if I enter any other character, the test succeeds (because `c` is not 'y'), the code block runs again and I am once again prompted to enter a character. And indeed, the loop continues executing until I enter 'y'.

In summary, the test condition comes at the *end* of a `do..while` loop but at the *beginning* of a `while` loop. This means that a `do..while` loop always executes at least once whereas a `while` loop may never execute at all if its condition evaluates to false the first time it is tested.

BREAK AND CONTINUE

There may occasionally be times when you want to break out of a loop right away – even if the loop condition does not evaluate to *false*. You can do this using the `break` statement. Just as `break` was used to make an immediate exit from a `switch..case` block, so too it will cause an immediate exit from a block of code that is run in a loop.

Look at the code that follows:

```
int i;
i = 0;
while( i < 10 ) {
    if( i == 5 ) {
          break;
    }
    printf("i = %d\n", i);
    i++;
}
```

The `while` loop has been set to run as long as `i` is less than 10. But inside that loop I test if `i` is 5 and, if so, I exit the loop with `break`. This means that in spite of the `while` condition expecting to run the loop ten times (for values of `i` between 0 and 9) in fact, when `i` is 5 the loop stops running. The code following `break` is not run so this is what is actually displayed:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Be careful when breaking out of a loop in this way. In a more complex loop, containing many lines of code, the `break` condition may not be obvious and it would be easy to assume that the loop would be guaranteed to run while `i` is less than 10. As a general principle, it is best (clearer and less bug-prone) to avoid breaking out of loops unless you have a very good reason for doing so.

Let me turn to a more useful example of a loop that uses a `break`. You will find this code in the *02_BreakAndContinue* project of Step 6. Here the `forbreak()` method 'encrypts' a string by adding 1 to the numeric (ASCII) value of each character in that string (`chararray`) and created a new string, `str`, from those encrypted characters:

02_BreakAndContinue

```
str[i] = chararray[i] + 1;
```

Suffice to say, this is about as simple (and insecure) as an encryption algorithm can get. Even so, it serves to illustrate the basics of how to process strings or arrays of other data-types using loops.

My code can handle strings up to 50 characters in length. In principle these might be strings entered at the keyboard or read from a file. For simplicity, however, I have created this string, as shown below, in order to illustrate how the program works:

```
char chararray[] = "Hello world! Goodbye";
```

As I want each string to have a maximum length of 50 characters, I set up a `for` loop to iterate over the characters from 0 to 49. However, the *actual* length of each string might be *less* than 50. So I have made it a requirement that each string is terminated by the `'!'` character. When that character is found the standard C string-terminator `'\0'` is substituted and the rest of the string is ignored because at that point I break right out of the `for` loop:

```
if (c == '!') {
    str[i] = '\0';
    break;
}
```

This means that if a string has only 10 characters ending with `'!'` the `for` loop will run only ten times (it will then exit on `break`) even though the loop is set up to run 50 times.

There is one other character that I want to treat specially. When a space is encountered – for example, the space character between the words "Hello world" I want to retain that space character rather than encrypt it. This is how my code does that:

```
if (c == ' ') {
    str[i] = c;
    continue;
}
```

The `continue` statement is a bit like a less dramatic version of `break`. Whereas `break` exits the loop and *stops* running the code in the loop any more, `continue` exits the loop but then *carries on* running the code in the loop.

So, in my example, if I `break` on `'!'`, the code in the loop stops running. Given the string `"Hello world! Goodbye"` the code would process up to the `'!'` character and no further. But the code will `continue` when it process the space `' '`. That means it exits the loop when `' '` is found after *"Hello"* but then it continues running the loop to process the rest of the characters, *"world"*.

As with `break`, you should use `continue` with caution. There may sometimes be perfectly good reasons for jumping out of code blocks using `break` or `continue` but jumps like this can make your code hard to understand and, in complex programs, they may result in subtle bugs.

This is another example of `break` and `continue` in the code that translates the encrypted string (by subtracting 1 from each character) back to the unencrypted version. This time, it copies the unencrypted characters back into the string, `str`, which initially contains the encrypted string. Since the space characters in that string are not encrypted, the code ignores spaces by executing `continue` when one is found. But it `break`s when the string terminator `'\0'` is found:

```
for( i = 0; i < 50; i++) {
c = str[i];
if (c == ' ') {
          continue;
    }
    if (c == '\0') {
          break;
    }

    str[i] = str[i] - 1;
}
```

This project also contains an example of breaking out of a `while` loop. This time it runs a `while` loop with no valid end condition (since `i` will always be greater than or equal to 0):

```
i = 0;
while( i >= 0 )
```

It displays the character and numeric code of each item in `chararrary` and it builds a new string that ends when the `'!'` character is found. A loop with no valid end condition will run forever unless you explicitly `break` out of it. In this loop, when `'!'` is found it breaks:

```
void whilebreak() {
    int i;
    char c;
    char str[50];
    i = 0;
    while( i >= 0 ){
          c = chararray[i];
          printf("[%d]='%c' ", i, c);
          if (c == '!'){
                str[i] = '\0';
                break;
          }
          str[i] = c;
          i++;
    }
    printf("\nAfter while loop, str='%s'", str);
}
```

MULTI-DIMENSIONAL ARRAYS

Before leaving the subject of arrays and loops, I want to mention multidimensional arrays. These are, in effect, arrays of arrays. You are not restricted to storing arrays that extend in single rows from 0 to some upper limit. You can also have arrays that have both rows and columns like a spreadsheet: one dimension stores the rows, the other dimension stores the columns (in fact, arrays can have more than two dimensions). Conceptually a two-dimensional array form a sort of 'grid' and you will find an example of this in the *03_MultiDimensionArrays* project.

Just as with single-dimensional arrays, I can initialize the elements of a multidimensional array but putting each element between curly brackets and assigning them to the array name at the time of its declaration. In this case, one array contains other arrays as its elements, so I need to initialize the arrays by placing one set of curly brackets (the 'outer array') to contain a comma-delimited list of arrays (the 'inner arrays') between curly brackets. These inner arrays contain comma-delimited lists of integers:

```
int grid [3][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15}
};
```

Here I have declared an array named `grid` that contains 3 arrays of 5 integers. You can think of this as a sort of matrix with 3 rows and 5 columns. If this were a spreadsheet, the intersection of each row and column would form a 'cell' and the contents of each cell would be an integer: 1, 2, 3, 4 and so on.

To access a specific piece of data I need to give its 'cell location' using square-bracket notation with the 'row' index in one pair of brackets and the 'column' index in another pair of brackets. This is how I would print out the value stored at row 1 and column 2 (remember arrays begin at index 0 so this gives the item on the second row in the third column) – namely, the integer 8.

```
printf("%d", grid[1][2]);
```

If you want to iterate over these arrays, you will need to iterate first over the 'rows' (the 'nested arrays') – for example, you could do this using a `for` loop like this:

```
for (row = 0; row < 3; row++)
```

This `for` loop initializes the `row` variable to the index of one of the 'nested' arrays on each turn through the loop. Now, in order to iterate over the items stored in each nested array (which I think of as the 'columns' of the grid), you would need to have another `for` loop inside the first `for` loop, like this:

```
for( column = 0; column < 5; column++ ) {
```

In my sample code, this is how I iterate over all the items stored in my `grid` two-dimensional array and display the ‘row’ number followed by the ‘column’ numbers and data (the `int`) stored at that location:

```
int row;
int column;
for (row = 0; row < 3; row++) {
    printf( "--- row %d --- \n", row);
    for( column = 0; column < 5; column++ ) {
          printf("column[%d], value=%d\n", column, grid[row][column]);
    }
}
```

This is what is displayed when the program runs:

```
--- row 0 ---
column[0], value=1
column[1], value=2
column[2], value=3
column[3], value=4
column[4], value=5
--- row 1 ---
column[0], value=6
column[1], value=7
column[2], value=8
column[3], value=9
column[4], value=10
--- row 2 ---
column[0], value=11
column[1], value=12
column[2], value=13
column[3], value=14
column[4], value=15
```

Now you might wonder what you would do if you wanted to break out of the inner `for` loop? Would the `break` exit just the inner loop or would it exit the outer loop too? Try it out. Here I add code to the inner loop that breaks when the value of `column` is 2:

```
for (row = 0; row < 3; row++) {
    printf( "--- row %d --- \n", row);
    for( column = 0; column < 5; column++ ) {
          printf("column[%d], value=%d\n", column, grid[row][column]);
          if (column == 2) {
                break;
          }
    }
}
```

This time, this is what I see when the program runs:

```
--- row 0 ---
column[0], value=1
column[1], value=2
column[2], value=3
--- row 1 ---
column[0], value=6
column[1], value=7
column[2], value=8
--- row 2 ---
column[0], value=11
column[1], value=12
column[2], value=13
```

From this you can see that the *inner* loop breaks when `row` is 0 and `column` is 2 but the *outer* loop continues running and it breaks again when `row` is 1 and `column` is 2 – and so on. This shows that `break` causes an exit from the innermost loop only. Once again, I should emphasize that breaking in this way is often not the most elegant way of terminating a loop. Whenever possible, try to terminate a loop only when the loop condition is met - for example when the test in a `while` loop or the expression in the second part of a `for` loop (e.g. `column < 5`) evaluates to false.

Chapter 7 – Strings and pointers

The most important thing to know about the string data-type in C is that there isn't one! Many other languages such as Java, C# and Pascal have a `string` type which lets you create variables to which string literals such as "Hello world" may be assigned.

Even though we have used strings many times in this course, you may have noticed that I have never created a variable of the `string` type – because there isn't one in C. Instead I have created strings like this:

01_StringIndex

```
char str1[] = "Hello";
```

Remember that a string is always terminated by a null `'\0'` character. It turns out that when you initialize a string at the time of its declaration, as in the example above, a null terminator is added automatically. The first null terminator found in a string will be treated as the end of that string. So, given this declaration:

```
char str2[] = "Goodbye\0 world";
```

When I display `str2` with `printf`, like this:

```
printf("%s\n", str2);
```

This is what is displayed (because the string terminates on the `'\0'` character):

```
Goodbye
```

Char Arrays and Pointers

In C, I can declare and initialize strings either by placing a pair of square brackets after an identifier or by preceding the identifier with an asterisk (or 'star') like this:

02_StringsAndPointers

```
char str1[] = "Hello";
char *str2 = "Goodbye";
```

At first sight, these two declarations appear to be more or less equivalent. Each is initialized with a string and I can display that string using `printf` like this:

```
printf("%s\n",  str1);
printf("%s\n",  str2);
```

In fact, the apparent similarity is deceptive. In order to understand why, we now have to get to grips with one of the most challenging aspects of the C language – pointers.

In the example above, the asterisk or 'star' (`*`) indicates that the variable `str2` is a pointer to some memory location. In this case, this happens to be the memory location where the array of characters forming the string "Goodbye" is stored. Each piece of data in your computer's memory is stored at some memory location or 'address'. You can display that address using the 'address-of' operator `&` placed before a variable name. This is how I would display the addresses of `str1` and `str2`:

```
printf("%d\n", &str1);
printf("%d\n", &str2);
```

If you ran this code, it would display some numbers such as:

```
2686746
2686740
```

These are the addresses – that is, the positions in your computer's memory where these variables live. Now that we have the addresses of the variables, let's take a look at their values – the data which they store. Here I will print out the address of each variable followed by its value shown first as an integer (`%d`) and then as a string (`%s`):

```
char str1[] = "Hello";
char *str2 = "Goodbye";
printf("%d %d %s\n", &str1, str1, str1);
printf("%d %d %s\n", &str2, str2, str2);
```

And this is what is displayed (though the actual numbers may vary):

```
2686746 2686746 Hello
2686740 4206628 Goodbye
```

This tells me that the address of `str1` is 2686746 and its value expressed as an integer is the *same* number 2686746. Its value expressed as a string is the string with which it was initialized, “Hello”.

The address of `str2` is 2686740 but its value expressed as an integer is a *different* number 4206628. Its value expressed as a string is the string with which it was initialized, “Goodbye”.

The important thing to observe here is that the value of the array name, `str1` when expressed as an integer is the *same* as the address of that name. In fact, we can say that:

> THE VALUE OF AN ARRAY **NAME** IS THE **ADDRESS** OF THE START OF THAT ARRAY.

But, unlike an array name, the value of the pointer variable, `str2`, is *not* its own address. It is a *different* address: it is the address at which the string “Goodbye” begins. So, in summary, this is what we can say about these two identifiers:

`1) char str1[] = "Hello";`

address of `str1` = `2686746`;
address of `"Hello"` = `2686746`;
value of `str1` as an integer = `2686746`;

`2) char *str2 = "Goodbye";`

address of `str2` = `2686740`;
address of `"Goodbye"` = `4206628`;
value of `str2` as an integer = `4206628`;

In other words, the *array* name, `str1` and the string “Hello” are one and the same thing and they indicate the same address:

Name	**str1**					
Data	**‘H’**	**‘e’**	**‘l’**	**‘l’**	**‘o’**	**‘\0’**
Address	`2686746`					

That is not true of the *pointer* variable `str2`. The string “Goodbye” occupies one address but the pointer variable `str2` occupies a different address. The data of `str2` is a number. And that number is the address of the string “Goodbye”:

Data	**‘G’**	**‘o’**	**‘o’**	**‘d’**	**‘b’**	**‘y’**	**‘e’**	**‘\0’**
Address	4206628 ✓							

Name	**str2**
Data	4206628
Address	2686740

So you can think of the pointer variable as pointing to the address of the string “Goodbye”.

> IN BRIEF: A char *array* name such as `char str1[]` is the actual memory location where an array of chars (such as “Hello”) is stored, whereas a char *pointer* variable such as `char *str2` is a number that represents the memory location where that array of chars is stored.

Since a char array name gives the address where a string begins, that means it must also give the address of the first character in that array. So where are the other characters in the char array located?

A char array, in common with any other array, is a series of items stored next to one another (contiguously) in memory. Each character in the array is located at a distance of one character’s worth of memory after the character that went before it.

The characters ‘H’, ‘e’, ‘l’, ‘l’, ‘o’ are placed one after the other in an array. That array starts with ‘H’. So we can think of the array as existing at a location in your computer’s memory that starts at the position or ‘address’ of the start of the array – that is, the address of the ‘H’ character.

For the sake of simplicity, let’s imagine that address of ‘H’ in this char array is 100. Let’s also imagine that the address of each subsequent character in the array goes up by 1. So the string “Hello” (terminating with the null `'\0'` character) can be thought of as living at these memory addresses:

‘H’	**‘e’**	**‘l’**	**‘l’**	**‘o’**	**‘\0’**
100	101	102	103	104	105

When I declare a char array, `str1`, like this…

```
char str1[] = "Hello";
```

…that name `str1` indicates the ‘address’ of the start of the array – here that would be the memory location, 100. My code can then treat each subsequent character in that array as a part of a complete string that only ends when a null character is found.

But what happens if I declare a char *pointer* variable like this?

```
char *str2 = "Hello";
```

This time the variable, `str2`, does *not* directly indicate the address of the start of the array. Instead it stores a *reference* to that address – that is, a number that corresponds to the position in the computer's memory of the char array "Hello". The number stored by a pointer variable is used to find the matching address. You could say, in fact, that the pointer variable ‘points’ to that address.

You can assign a char pointer variable to the address of an array (given by the array name) using the address-of operator `&`. So, given the array `str1` and the array-pointer `str2`:

03_StringsAndAddresses

```
char str1[] = "Hello";
char *str2 = "Goodbye";
```

You can make `str2` point to `str1` using the `&` operator like this:

```
str2 = &str1;
```

Or you can make `str1` point to `str2` using the array name (without the `&` operator) like this:

```
str2 = str1;
```

The end result is the same. In both examples above, following the assignment, `str2` points to the address of the array `str1` because the array name itself, `str1`, gives the address of the array.

Strictly speaking an array name is not a variable. The value of an array name – for example, the address of a char array such as str1 – is calculated when you compile your program. Unlike a variable, an array name cannot be assigned a new value when your program is running (though new values can be assigned to its individual elements).

STRINGS AND FUNCTIONS

You can pass a string to a function using either the character-array (square brackets) or character-pointer syntax (*). This is valid:

04_ReturnStrings

```
void string_function(char astring[])
```

And so is this:

```
void string_function(char *astring)
```

To return a string, declare the function return-type as char *. Here is an example of returning a string with a maximum length of 100 characters, defined by MAXSTRLEN (note that it is the programmer's responsibility to ensure that this length is not exceeded):

```
#define MAXSTRLEN 100

char greeting[MAXSTRLEN];

char * string_function(char astring[]) {
    strcat(greeting, "hello ");
    strcat(greeting, astring);
    strcat(greeting, "\n");
    return greeting;
}
```

Here the string to be returned is created using the strcat function which concatenates (adds on) one string with another. In this case, the greeting array that is returned from the function has been declared outside of the function itself to ensure that it continues to exist outside that function. In other words, greeting is within the 'global scope' of my entire program. I'll explain scope in more detail in the next chapter. I can call this function like this:

```
printf(string_function("Fred"));
```

The result is that the functions return value "hello Fred\n" is displayed by `printf()`. But what if I want to return a string without first declaring a 'global' array such as `greeting`? It turns out that's a bit more complicated. In order to do this safely, I have to set aside a bit of memory to hold the string. I do this using the `malloc` (memory allocation) function which allocates space for my string on a global memory area called the 'heap'.

THE STACK AND THE HEAP

The local variables declared inside a function are allocated memory in an area known as the stack. When you exit from a function, the variables on the stack are cleaned up. To all intents and purposes, they cease to exist. The `malloc()` function allocates space in a different part of your computer's memory called the 'heap'. The data assigned to variables that are allocated on the heap continues to exist even after you've exited the function in which they were allocated.

I have to tell `malloc` how much memory – that is, how many 'bytes' – I need to be allocated. It doesn't matter too much if I allocate more memory than I really need but it could be catastrophic if I allocate less memory than I need. In this case, I have decided to allocate 100 bytes (`MAXSTRLEN`), which is enough for 100 normal characters:

```
char * string_function_dynamic(char *astring) {
    char *s;
    s = malloc(MAXSTRLEN);
    s[0] = 0;
    strcat(s, "hello ");
    strcat(s, astring);
    strcat(s, "\n");
    return s;
}
```

Note that I have had to initialize the array `s` with a null character (assigning the integer 0 has the same effect as assigning the null char `'\0'`) before attempt to concatenate another string such as "hello" onto it.

It is good practice to free memory when no longer required. In small programs failure to free memory is unlikely to cause any problems. In large programs it might. Freeing memory is discussed in *Chapter 10*.

CASTS

Some compilers may not permit this assignment:

```
s = malloc(MAXSTRLEN);
```

That is because `malloc()` returns a pointer to `void` and (strictly speaking) that is not compatible with an array of characters. To ensure compatibility, it is best to 'cast' the returned pointer to a character pointer. A 'cast' is an instruction to treat compatible types as equivalent. When I cat one type to another type I am, in effect, telling the compiler to 'shut up' and stop complaining. A cast is done by placing the intended type between parentheses prior to the variable or expression that returns a value. This is how I cast the return pointer from `malloc` to a `char` pointer:

```
s = (char*)malloc(MAXSTRLEN);
```

STRING FUNCTIONS

There are various useful string functions in the standard libraries provided with your C compiler. You should include `string.h` in order to use these string functions:

```
  #include <string.h>
```

Just about all C compilers provide functions to do common operators such as concatenate and copy strings. Many C compilers also provide additional functions and you should be sure to read the documentation for your C compiler to check on the full range of functions available.

Here I am going to show just a few of the traditional functions for use with strings and characters. These functions should be available with all commonly-used C compilers.

What is the difference between a string and a character?

Be careful when you enter string and character 'literals' – that is, actual bits of string and character data such as "Hello world", "x" or 'x'. A C string literal is placed between two double-quotes whereas a char is placed between two single quotes. Some functions that expect to operate on a char will, in fact, allow you to enter a string. But remember that a string – even one such as "x" that contains a single character – is really a memory location at which an array of characters begins. This means that if you accidentally enter the string "x" when you had intended to enter the char 'x' your code may not produce the expected results. Look at this code which uses the isalpha() function to test if a char is an alphabetic character:

```
if(isalpha('x')){
 printf("'x' is a letter");
}else{
 printf("'x' is not a letter");
}
```

When run, this displays:

```
'x' is a letter
```

But what happens if I accidentally enter the string "x" instead of the char 'x':

```
if(isalpha("x"))
```

In fact, this time, the program displays:

```
'x' is a not letter
```

Commonly used string functions include:

05_StringFunctions

`strlen()` to find the length of a string.

Example:

```
strlen("Hello")
```

Output:

```
5
```

`strcat()` to concatenate two strings or `strncat()` to concatenate a specified number of chars from one string to another.

Example: Given a string, `msg`, which contains the string "Result: ".

```
strcat(msg, "Easter");
printf("\n\nstrcat: '%s'", msg);
```

Output:

```
Result: Easter
```

```
strncat(msg, "Easter", 4);
printf("\nstrncat: '%s'", msg);
```

Output:

```
Result: East
```

NOTE. When copying or concatenating strings, you need to ensure that the destination string has enough memory to hold all the characters after the operation., In my code, I declare the destination strings with a fixed length like this:

```
#define MAXSTRLEN 200
char msg1[MAXSTRLEN] = "Result1: ";
```

`strncpy()` to copy one string to another string

Example: Given an empty string, `myotherstr`.

```
strncpy(myotherstr, "Easter", 4);
printf("\nCopied this string: '%s'", myotherstr);
```

Output:

```
Copied this string: 'East'
```

`strstr()` to search for a substring (returned as a pointer) in a string

Example:

```
char *ptrtostr;
int foundat;
char str[] = "abcedfg";
ptrtostr = strstr( str, "ced" );
foundat = (int)((ptrtostr - str) + 1);
if ( ptrtostr != NULL ){
      printf( "\nFound at position %d\n", foundat );
}
```

Output:

```
Found at position 3
```

C11 String functions

Some modern C compilers which support the 'C11' standard of the language provide a number of safer alternatives to the traditional C functions. These function names end with `_s`. These functions typically take additional arguments that help verify that only a fixed number of characters are used when, for example, copying or concatenating strings. This can help to avoid problems such as buffer overruns (as I discussed in Chapter 4). Here is an example of using `strncpy_s` to copy the string "Easter" to the `char` array `mystr` (defined as `char mystr[6]`). The 2nd argument defines the size of the destination string and the fourth argument defines the number of characters to copy – the end result is that `mystr` is assigned the characters: "East".

```
strncpy_s(mystr, 6, "Easter", 4);
```

At the time of writing, not all C compilers provide these functions.

CHAR FUNCTIONS

There are also some functions intended for use with single characters. To use these you will need to include `ctype.h`. These are the most common `char` functions. Each takes a `char` argument and return a non-zero value representing *true* or a 0 value if *false*:

`isalnum()`	An alphabetic letter or a digit
`isalpha()`	An alphabetic letter
`isblank()`	A space or tab character
`iscntrl()`	A control character
`isdigit()`	A digit
`isgraph()`	Any printing character except a space
`islower()`	A lowercase letter
`isprint()`	Any printing character including a space
`ispunct()`	A printing character that is not a space or alphanumeric
`isspace()`	A whitespace character (‘ ‘, ‘\n’, ‘\t’, ‘\v’, ‘\r’, ‘\f’)
`isupper()`	An uppercase letter
`isxdigit()`	A hexadecimal digit

You can use these functions to test characters. The following example counts the occurrences of upper and lowercase characters, digits, blanks and punctuation:

06_CharFunctions

```
char mystring[] = "On the 2nd day of Christmas my true love gave to me,
2 turtle doves and a partridge in a pear tree.";

void chartypes() {
    int i;
    char c;
    int numDigits = 0;
    int numLetters = 0;
    int numUpCase = 0;
    int numLowCase = 0;
    int numSpaces = 0;
    int numPunct = 0;
    int numUnknown = 0;
    int lengthOfStr;
    lengthOfStr = strlen(mystring);

    for (i = 0; i < lengthOfStr; i++ ) {
          c = mystring[i];
          if(isalpha(c)) {
                numLetters++;
                if (isupper(c)) {
                      printf("'%c' [uppercase character]\n", c);
                      numUpCase++;
                } else {
                      printf("'%c' [lowercase character]\n", c);
                      numLowCase++;
                }
          } else if (isdigit(c)) {
                printf("'%c' [digit]\n", c);
                numDigits++;
          } else if (ispunct(c)) {
                printf("'%c' [punctuation]\n", c);
                numPunct++;
          } else if (isblank(c)) {
                printf("'%c' [blank]\n", c);
                numSpaces++;
          } else {
                printf("'%c' [unknown]\n", c);
                numUnknown++;
          }
    }
 printf("This string contains %d characters: %d letters
          (%d uppercase, %d lowercase)\n",  lengthOfStr, numLetters,
          numUpCase, numLowCase);
 printf( "%d digits, %d punctuation characters, %d spaces and %d
       unclassified characters.\n", numDigits, numPunct, numSpaces,
       numUnknown );
}

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

Chapter 8 – User defined types and scope

Let's suppose you want to create a program that stores a 'database' of your audio CD collection. Each CD would need to include the following data items: the CD name, the recording artist, the number of tracks on the CD and a user-rating.

STRUCTS

The problem is that, unlike `char` or an `int`, a `cd` is not a known data type. As it turns out, this is not a major problem since you can create user-defined data-types in the form of structures or `struct`s. Here is a `struct` capable of holding all the data items for a CD:

01_Structs

```
struct cd {
    char name[50];
    char artist[50];
    int trackcount;
    int rating;
};
```

To declare a `struct`, begin with the keyword `struct` followed by a pair of curly brackets. Between the curly brackets, put a semicolon-delimited list of one or more variables. The closing curly bracket should be followed by a semicolon.

Once a named `struct` has been declared you can create variables of the *type* of that named `struct`. This is how you might declare a simple `cd` variable called `thiscd` and a ten-slot `cd` array variable called `cd_collection`:

```
struct cd thiscd;
struct cd cd_collection[10];
```

To declare a `struct` variable, begin with the keyword `struct` followed by the `struct` name (such as `cd`, which you defined earlier), then a variable name and a semicolon.

A `struct` definition (such as `cd`) acts as a blueprint from which multiple `struct` variables (possibly called `mycd`, `yourcd`, `anothercd` and so on) may be created. You can think of each `struct` as a data 'record' and each of its variables as 'fields' in that record. To access the fields you just enter the `struct` variable name, then a dot, then the field name like this: `thiscd.name`, `thiscd.artist`.

For example, let's assume you have created an array of `cd struct`s called `cd_collection`, from 0 to an upper limit defined by the constant `NUMBER_OF_CDS`. This is how you might write a loop to display the data from each field of each `struct` in the array:

```
int i;
struct cd thiscd;
for (i = 0; i < NUMBER_OF_CDS; i++) {
    thiscd = cd_collection[i];
    printf("CD #%d: '%s' by %s has %d tracks. My rating = %d\n",
           i, thiscd.name, thiscd.artist,
           thiscd.trackcount, thiscd.rating);
}
```

Just as you can access the data fields using the field name after a dot, so too you can assign values to those fields using dot-notation like this:

```
thiscd.trackcount = 20;
```

If the `struct`s are stored in an array, you can access each element of the array and assign values like this:

```
cd_collection[0].trackcount = 20;
```

In the sample program **01_Structs**, this is how I assign data to the fields of four `cd` records stored in the `cd_collection` array:

```
strcpy(cd_collection[0].name, "Great Hits");
strcpy(cd_collection[0].artist, "Polly Darton");
cd_collection[0].trackcount = 20;
cd_collection[0].rating = 10;

strcpy(cd_collection[1].name, "Mega Songs");
strcpy(cd_collection[1].artist, "Lady Googoo");
cd_collection[1].trackcount = 18;
cd_collection[1].rating = 7;

strcpy(cd_collection[2].name, "The Best Ones");
strcpy(cd_collection[2].artist, "The Warthogs");
cd_collection[2].trackcount = 24;
cd_collection[2].rating = 4;

strcpy(cd_collection[3].name, "Songs From The Shows");
strcpy(cd_collection[3].artist, "The Singing Swingers");
cd_collection[3].trackcount = 22;
cd_collection[3].rating = 9;
```

Typedefs

In the last example, I created a new data-type defined by a `struct` which I called `cd`. But, unlike standard data types such as `char` and `int` I was not able to create new `cd` variables just by proceeding the variable name with the type name. So this was not allowed:

```
cd thiscd;
```

Instead I had to precede the variable declaration with the keyword `struct`, like this:

```
struct cd thiscd;
```

In fact, there is a way of creating new types that allow the declaration of variables using the same syntax as you would use for standard types. To do this you must explicitly define a type using the keyword `typedef`. By tradition it is normal to name types with an initial capital such as `Mytype` or `Yourtype`. This is how I declare a type named `CD` (in uppercase) which identifies my `struct` named `cd` (in lowercase):

02_Typedefs

```
typedef struct cd CD;

struct cd {
    char name[50];
    char artist[50];
    int trackcount;
    int rating;
};
```

Alternatively, you can combine the `typedef` with the `struct` declaration like this (where the `typedef` name is placed after the closing curly bracket):

```
typedef struct cd {
    Str50 name;
    Str50 artist;
    int trackcount;
    int rating;
} CD;
```

Having done this I can now create `CD` variables like this:

```
CD thiscd;
```

I can declare an array of CD records like this:

```
CD cd_collection[NUMBER_OF_CDS];
```

You can also typedef standard types such as int or char. In principle, this may provide an 'alias' in the interests of readability. In my sample program I have provided an alias, Str50, for a 50-element char array:

```
typedef char Str50[50];
```

As you can see, I have used this alias when declaring the string fields of my cd struct:

```
Str50 name;
Str50 artist;
```

ENUMS

Sometimes your programs may need to work with a fixed set of related values – for example, a calendar program may need to deal with the seven days of the week, whereas a gambling program might deal with four suits of playing cards. In order to represent these values you could use integers from 0 to 6 for days or from 0 to 3 for suits of cards respectively. But often it would be clearer if you could use named identifiers instead of numbers. For example, consider this code:

```
if ((card == 0) || (card == 1)) {
    printf("This card is red.\n");
```

If you use named identifiers, the above code could be rewritten more clearly, like this:

```
if ((card == Hearts) || (card == Diamonds)) {
    printf("This card is red.\n");
```

There is a simple way to create series of identifying names such as Hearts, Diamonds, Spaces and Clubs. You do so by creating an 'enumerated type' or enum. This is how I would define the four suits of a deck of cards:

```
enum suits {
    Hearts, Diamonds, Clubs, Spades
};
```

Now, when I want a variable of the `suits` 'type' I can create it like this:

```
enum suits playingcard;
```

Similarly, I could create an `enum` of the days of the week like this:

```
enum days {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
};
```

And I could create a variable of that type like this:

```
enum days today;
```

When I want to assign a value to that variable, I can use one of the identifiers listed in the `enum` definition like this:

```
today = Saturday;
```

ENUMERATION CONSTANTS

The identifiers listed in an `enum` are actually *constants* because their values cannot be changed. In fact, an `enum` may sometimes be described as an 'enumeration constant'.

By default, the identifiers in an `enum` are assigned integer values from 0 upwards. So in my `days enum`, `Monday` has the value 0, `Tuesday` has the value 1 and so on. In fact, it is also possible to assign plain integer values to `enum` variables (though doing so would defeat the object of the exercise).

Note that not only can you assign simple integers to an `enum` variable instead of using the identifiers in the `enum`, but you may even assign integers beyond the scope of those declared in the `enum`. For example, if you have four `enum` constants (with values from 0 to 3) representing a suit of cards, you may be able to assign some other value to a variable of that `enum` type like this: `playingcard = 100`. Some C compilers or development environments may warn you if you attempt to do this but that is not guaranteed!

You can also, optionally, assign specific values to each identifier in an `enum` at the time of declaration. Here, for example, I assign numerical values that correspond to the name of each constant:

```
enum numbers {
    Couple = 2, Dozen = 12, Score = 20
};
```

If you want to pass `enum` values to functions, be sure to declare the function argument appropriately. For example, here is a function that expects an argument, `num`, of the `enum` type called `numbers`:

```
void buyinbulk(enum numbers num) {
   printf("The customer wants to buy %d items.\n", num);
}
```

I have rewritten my CD Database program using an `enum` to represent the user rating of each `CD` record. This is my `enum`:

03_Enums

```
enum score {
    Terrible, Bad, Average, Good, Excellent
};
```

This is my `CD struct` definition (note the `enum score` variable):

```
typedef struct cd {
   Str50 name;
   Str50 artist;
   int trackcount;
   enum score rating;
} CD;
```

And this is an example of the assignment of a `rating` to the first `CD` record in the `cd_collection` array:

```
cd_collection[0].rating = Excellent;
```

Header Files

Real-world C programs are typically composed of many separate source-code files. Even a simple "Hello world" program such as the very first program from Step 2 of this course, makes use of code from more than one file. I entered all of my own code into a single file. But that file imported code from another file that was provided with my C compiler, like this:

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

That tells the compiler to 'include' (that is, to make available to my program) the contents of the file named `stdio.h`. This is one of the standard 'library' files supplied with C compilers. The angle brackets < > tell the compiler to search for this file in the location reserved for the standard library files. When the file is found its contents are inserted (just as though you had copied and pasted them) at the point in your source file where it is included.

> To include header files supplied as standard with a C compiler, use angle brackets like this:
>
> ```
> #include <string.h>
> ```
>
> To include header files from the current directory – that is, files that you have created within the current project, use double-quotes like this:
>
> ```
> #include "mystring.h"
> ```

A file with the extension '.h' is called a 'header file'. A header file typically contains definitions of functions and constants. The header file does not contain the *implementation* of functions – only their declarations. The implementations are generally contained in source code files that end with the extension '.c'.

While you should always include the header files that define the functions and constants you need, you may be surprised to discover that sometimes you will be able to use functions which are declared in header files even if you forget to include those header files. This is because the C compiler may have compiled the C source code files containing the *implementation* of the functions. However, if you don't include the header file that *defines* those functions, the compiler may not have been able to check that all the data-types used by the function-calls in your program – that is, the types of the data returned from functions or passed to functions as arguments – are correct.

Header Files and the C compiler

To understand why header files are important, you need to understand how your C source code files are translated into executable programs. Essentially this is done in three steps:

Stage 1: First your files are preprocessed. The preprocessor deals with all the special commands preceded by a hash `#` character – such as `#define` to define constants and `#include` to include header files.

Stage 2: Then all the C source-code files in your program are compiled. The compiler translates the source code into an intermediate code format called 'object code' and this object code is saved into a number of separately compiled files.

Stage 3: Finally, all the separately compiled files are combined in a process called 'linking'. This is when the final executable program is created. This executable program contains 'machine code' – that is, code which is capable of being run by your computer hardware.

Bear in mind that at *Stage 2* - the compilation stage - the compiler processes each source code file *separately*. You might have one file that *contains* a function called `xxx()` and nine other files that *call* the `xxx()` function. Since the compiler processes each file one at a time, it cannot verify that the implementation of `xxx()` in *one* file matches the function-calls to `xxx()` in *all the other* files. The file *containing* the `xxx()` function and the files *calling* the `xxx()` function are only put together in the final stage – by the linker. By that time, if there are any problems it's too late to fix them.

But if you include a header file which defines exactly what types of arguments the `xxx()`function takes and what data-type, if any, it returns, the compiler will be able to refer to the definition in the header file. When it compiles each C source code file it can verify that they all use exactly the same definition of `xxx()` (as specified in the header file). If any file does not use the correct definition, the compiler is able to spot the problem and show a warning message.

> Put simply, when you define functions in a header file, that file can be used by the compiler as a sort of reference document. As each individual source code file is compiled, the compiler refers to the definitions in the header file to make sure that all the source code files agree on the types of data sent to and returned from the functions they use. The declarations of functions in a header file help the compiler to find potential problems before they become bugs.

04_HeaderFiles

In my sample project, you will see that I have created a source-code file called **mystring.c**, which contains the `readln()` function which I wrote in Chapter 4 and a version of the **searchstring()** function from Chapter 7. By placing these functions in their own file, it makes it easy for me to reuse them in different projects. The header file **mystring.h**, declares these functions plus a defined constant, `BUFSIZE`:

```
#define BUFSIZE 100
int readln(char[], int);
int searchstring(char[], char[]);
```

Notice that it is not necessary to provide parameter names in the function declarations in a header file – only their types such as `char[]` and `int`. The **main.c** file includes this header file like this:

```
#include "mystring.h"
```

This file now has access not only to the functions declared in **mystring.h** but also to the `BUFSIZE` constant:

```
char b[BUFSIZE];
```

SCOPE

'Scope' describes the 'visibility' of the variables, functions and constants in your code. Every program is made up of elements with well-defined boundaries. These program elements may be anything from an entire code file to a single function – and each of these elements defines a 'scope'.

LOCAL VARIABLES

Variables that are defined inside a function are said to have 'local' scope. They only 'exist' within the function itself. Code outside the function cannot access local variables. In the example below, the variable `num` is local to `func1()` and so it cannot be used in `func2()`:

```
void func1(){
    int num;
    num = 100;
}

void func2(){
    num = 200;   // This is an error!
}
```

The parameters declared between parentheses in a function header also have local scope within that function.

GLOBAL VARIABLES

Variables that are defined outside a function (that is, in the main 'body' of the code file) are said to have 'global' scope. They are available to all the functions in the code file. In the example below, the variable `num` is global so both `func1()` and `func2()` can use it:

```
int num;

void func1(){
    num = 100;
}

void func2(){
    num = 200;
}
```

Note that when a local variable is declared with the same name as a global variable, the local variable will be used instead of the global variable.

```
int num = 50;

void func1(){
    num = 100;
 printf( " %d\n", num );    // prints local variable value: 100
}

int main(int argc, char **argv) {
    func1();
 printf( " %d\n", num );    // prints global variable value: 50
}
```

SCOPE AND EXTERNAL FILES

The functions declared in source code files throughout a project are, by default, accessible by all the files in that project. As mentioned before, it is good practice to declare functions to be share among multiple source files in a header file. However, the header file declarations do not themselves change the scope of the external files. Even if they are not declared in the header file, the external functions may still be 'in scope' throughout your project.

> While the name of a header file (such as **mylib.h** with the extension **.h**) often matches the name of a C source-code file (such as **mylib.c** with the extension **.c**) this one-to-one matching between header and source-code file is not obligatory. For example, in the sample project *05_Scope*, I have one header file, **mystring.h**, which declares functions found in two source-code files, **mystring.c** and **mystringutils.c**.

STATIC FUNCTIONS

Sometimes you may want to make sure that the functions inside one file cannot be used by the functions in some other file. You can do that by declaring the functions using the `static` keyword. This ensures that the functions remain *private* – that is, that they are only accessible within the file in which they occur.

05_Scope

In the *05_Scope* project I have declared the `searchstring()` function to be `static` inside the file **mystringutils.c**:

```
static int searchstring(char searchstr[], char sourcestr[]) {
```

I have made this function static because I want it to be available only to the function named `findsubstring()` within the same code file. It is my intention that when someone wants to search for one string inside another they must call my `findsubstring()` function. The `searchstring()` function is only ever intended to be used by the `findsubstring()` function and I want to be absolutely sure that nobody can call `searchstring()` from some other code file. By making the function `static` I have, in effect, hidden it from external code files.

When declared outside a function, static variables have the same (file-level) scope as static functions.

LOCAL STATIC VARIABLES

Confusingly the keyword `static` has a different meaning when it is applied to variables declared within a function. A `static` *function* is private within the file in which it occurs. But a `static` *variable* which is declared *inside* a function is one that retains its value between function calls.

Let’s suppose you have a `static int` variable called `x` inside a function called `addnumbers()` and `x` is initially assigned the value 0. Inside the function some code increments the value of `x` by 1 each time the function is called. If the function is called three times, `x` will now have the value 3.

```
void addnumbers(){
    static int x = 0;
    int y = 0;
    x++;
    y++;
    printf("In addnumbers() x=%d, y=%d\n", x, y );
}

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

This is the output that will be displayed when the program above is run:

```
In addnumbers() x=1, y=1
In addnumbers() x=2, y=1
In addnumbers() x=3, y=1
```

Most programs are unlikely to need to use local `static` variables. But while local `static` *variables* are somewhat esoteric, `static` *functions* are not. It is highly likely that programs of any complexity will want to make some functions private (that is, `static`) so that they cannot accidentally be used by other files.

Chapter 9 – Files

Up to now, my CD database suffers from one enormous limitation: the records are 'hard-coded' into the application itself so there is no way to save new records to disk and reload them into memory later on. In order to be able to do that, I need to know how to create files capable of storing all the data from my list of CD records. I'll get around to that in Chapter 10. First, though, I'll explain the basics of using files in C. In this chapter I will be working with plain human-readable text files. Even so, some operations are common to all files, no matter what type of data they contain.

Opening and closing files

A file on disk is a storage area containing data which may either be represented as sequences of printable characters – a 'text file' – or else may be some other kind of digital representation of formatted data – a 'binary file'.

Before you can use a file in your program, you must 'open' it. A file may be opened for reading – when you want to load data into memory, for writing – when you want to save data from memory back into the file or for reading and writing when you want to be able both to load and save data. When you have finished using the file you should 'close' it.

To open a file you can use the `fopen()` function. This takes the file name as its first argument, and some characters specifying the file access mode as the second argument.

> Your compiler may support alternative file-handling functions such as `fopen_s()` which is more secure than `fopen()`. Check the compiler documentation to see which additional functions are available. Not all compilers support these alternative functions.

The file access mode lets you open a file for reading (`"r"`), writing (`"w"`) or appending to a text file (`"a"`). You can also open a file for reading and writing (`"r+"`) for a file that already exists, or destructive reading and writing (`"w+"`) which means that the file is truncated to zero length – effectively its contents are deleted, when it is opened. When handling binary files you need to add the character '`b`' to these modes (`"rb"`, `"wb"` and so on). For the full range of supported file access modes, refer to your compiler's documentation.

File Access Mode Summary

`"r"` Opens for reading. If the file does not exist or cannot be found, the call to `fopen()` fails and 0 (or the predefined constant `NULL`) is returned.

`"w"` Opens an empty file for writing. If the given file exists, its contents are destroyed.

`"a"` Opens for writing at the end of the file (by appending). Creates the file if it does not exist.

`"r+"` Opens for both reading and writing. The file must exist.

`"w+"` Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.

`"a+"` Opens for reading and appending. Creates the file if it does not exist.

When you open a file using `fopen()`, the function returns a 'file pointer'. You may then use that pointer with various functions to open and close the file or read and write data to and from it. If the file cannot be opened, 0 (or the predefined constant `NULL`) is returned.

File Pointers

A File pointer is not a pointer directly into the file on disk. In C, **FILE** is a predefined structure that contains information about a file, such as the file mode and the current position in a file (when characters are being written or read to and from it) and when the end of the file is reached. A file pointer points to a **FILE** structure.

TEXT FILES

You will find some examples of basic file operations in the **01_FileBasics** sample program, which opens and closes a text file, "*test.txt*", reads and writes lines of text to and from the file. This program also shows ways of deleting the file contents and removing the file itself.

01_FileBasics

First I define constants to specify the maximum length of a line of text and the name of the file to be used in my code:

```
#define MAXSTRLEN 200
#define FILENAME "test.txt"
```

This function opens the file for writing. If the file does not exist, it is created. If it does exist, its contents are deleted before any data is written. Two lines of text are then saved into the file and the file is then closed:

```
void writelines() {
    FILE *fp;
    fp = fopen(FILENAME, "w");
    fputs("Hello world\n", fp);
    fputs("Goodbye Mars\n", fp);
    fclose(fp);
}
```

If the specified file can be opened, this function reads lines of text from the file while there are still more lines to be read and displays those lines on screen (`stdout`), closing the file when all lines have been processed. If the file cannot be opened (in which case the file pointer `fp` is 0), an error message is displayed:

```
void readlines() {
    FILE *fp = fopen ( FILENAME, "r" );
    char line [MAXSTRLEN];
    if ( fp != 0 ) {
          while ( fgets ( line, sizeof(line), fp ) != 0 ) {
                fputs ( line, stdout );
          }
          fclose ( fp );
    } else {
          printf( "File %s cannot be opened!", FILENAME);
    }
}
```

This function opens a file for writing but does not write any data into it. Opening a file for writing has the effect of deleting the file contents, so this function erases the contents of the file but leaves the file itself present on disk:

```
void clearfile() {
    FILE *fp;
    fp = fopen(FILENAME, "w");
    fclose(fp);
}
```

If you want to remove the file itself rather than just delete its contents, use the remove() function. This returns 0 if successful.

```
void deletefile() {
    if( remove(FILENAME) == 0 ) {
          printf("%s file deleted.\n", FILENAME);
    } else {
          printf("Unable to delete the file: %s.\n", FILENAME);
    }
}
```

Since I am reading one line at a time from a text file, it would be very simple to write a program that counts the number of lines that the file contains.

This is how I've done that:

02_FileStats

```
void linecount(char *fn) {
    int numlines = 0;
    char line[MAXSTRLEN];
    FILE *fp = fopen ( fn, "r" );
    if ( fp != 0 ) {

          while ( fgets ( line, sizeof(line), fp ) != 0 ) {
                numlines++;
          }
          fclose ( fp );
          printf("%s contains %d lines of text.\n", fn, numlines);
    } else {
          printf( "File %s cannot be opened!\n", fn);
    }
}
```

If can pass to this function the name of a text file containing some text, it displays the number of lines. If I pass the name of an empty file it tells me that the file contains 0 lines. And if I pass the name of a file that does not exist, it tells me that the file cannot be opened.

As this program reads one line of text at a time, I can treat those lines of text in the same way that I might treat lines that were entered by the user at the keyboard. For example, you may recall that in a previous program I wrote a function to search for substrings in a string entered by the user. I can use that same function to search for substrings in each line read from a disk file. Here is a simple program that calculate the number of lines which contain a specified substring in a file:

03_FileSearchStr

```
static int searchstring(char searchstr[], char sourcestr[]) {
    char *ptrtostr;
    int foundat;
    foundat = -1;
    ptrtostr = strstr( sourcestr, searchstr );
    if ( ptrtostr != 0 ) {
          foundat = (int)((ptrtostr - sourcestr));
    }
    return foundat;
}

void findstrings(char *fileName, char *ss) {
    FILE *f;
    int count;
    char line[MAXSTRLEN];

    f = fopen(fileName, "r");           // open file read only
    if (f == 0) {
          printf("Can't open the file: '%s'\n", FILENAME);
    } else {
          count = 0;                    // initialize the count
          while ( fgets ( line, MAXSTRLEN, f ) != 0 ) {
                if (searchstring(ss, line) >= 0) {
                      count++;
                }
          }
          printf ("'%s' was found in %d lines\n", ss, count);
          fclose(f);                    // close it
    }
}
```

You can try adding your own functions. For example, if you read the text in one character at a time instead of one line at a time, you could count the characters or change uppercase characters to lowercase. Or you could encrypt the text as we did back in Chapter 6.

This is all very well if the files you are using contain nothing but plain text. But how do you read and write more complex data such as the multi-field CD records in my CD database? That is the subject of the next chapter.

Chapter 10 – Binary Files and Memory management

In this final chapter I want to consider a few of the special features – and problems – associated with binary files. When I talk about 'binary files' I mean any file that contains data which cannot be treated as plain 'readable' text. Often this may be a file that stored some sort of specially formatted data such as, for example, the data of the records (that is, the `struct`s) in my CD database. Sometimes files of this sort are called 'random access' files because I may decide to access records at specific locations (these locations are not really 'random' in spite of the name!) rather than reading data sequentially from the start to the end of the file.

Of course, all files – even text files – are binary because all files contain binary data which ultimately is made up from binary digits or 'bits'. But text files are so widely used that it is often convenient to treat them specially – in particular by assuming that the file contains readable characters arranged on lines that end with linefeeds.

But you can, if you wish, treat even a text file as a binary file. For instance, in the *01_ReadFile* project supplied with this step, I open a text file in binary reading mode:

01_ReadFile

```
f = fopen(fileName, "rb");
```

Now, instead of processing its contents one line at a time, I read the entire file into memory and process it one character at a time. Essentially, I fill an array (a 'buffer') `b` with the contents of the file, iterate over the characters and increment a variable, `linecount`, each time the newline '`\n`' character is found. But that isn't the interesting part of this program. The really interesting element is how I calculate the size of the buffer needed to hold the contents of the file. After all, I don't know in advance how big the file will be so I can't declare a fixed-size array in my code. Let's look at how this is done in detail.

Bear in mind that, when I open a file with `fopen()` the function returns a file pointer. My pointer is the variable, `f`. I can now use various functions to make this pointer point to various positions in the file.

Incidentally, while it is commonly said that file pointers 'move' to or 'point' to positions in a file, it would be more accurate to say that the file structure (to which a file pointer points) refers to a new position in a file.

Moving a file pointer to a specified position is called seeking and my code starts by using the `fseek()` function to seek to the end of the file. I do this using the `SEEK_END` constant as the third argument:

```
fseek(f , 0 , SEEK_END);
```

Now I can use the `ftell()` function to find the position of the file pointer. This position is given in bytes – with one byte per character – so the position here tells me how many characters are in the file and I assign that value to the `size` variable:

```
size = ftell(f);
```

Now I want to move the pointer back to the start of the file. The `rewind()` function does that:

```
rewind(f);
```

Memory Allocation

Finally I am ready to create a buffer – an array – capable of holding the number of characters in the file. To do this I need to create an array with enough space to hold the characters. I do this using the `malloc()` function to allocate `size` amount of memory and assign this to `b`, like this:

```
b = (char*)malloc(size);
```

> The `malloc()` function returns a pointer to `void` so I have had to cast this to a pointer to a character `(char*)`. Not all compilers require this cast but it is, in any case, good practice. For more information on type-casts of this sort, refer back to the section on casts in *Chapter 7*.

Now I can fill this array with characters by reading the contents of the file using `fread()`:

```
items_read = fread(b, 1, size, f);
```

The arguments to `fread()` are: 1) `b` – the buffer or array into which the data will be read, 2) the size of each data item to read – here that's `1` for a single `char`, 3) `size` – which is the number of items (here the number of characters) to be read and 4) `f` - the file pointer to the source data file. The returned value, `items_read`, gives the number of items that were read.

INTEGER TYPES

You may notice that the `items_read` variable is declared as being of the `size_t` type:

```
size_t items_read;
```

It turns out that `size_t` is a defined type – that is, the C library code has used a `typedef` to define `size_t` to be a `long unsigned int`. The C language recognises various integer types, some of which are 'signed' (they may have both plus and minus values), some of which are unsigned so cannot have negative values - and some of which have different upper and lower values. For example, a `long` may be capable of holding a bigger range of values than an `int`, though these value ranges vary according to the compiler. In most cases, a simple `int` will suffice. However, as the C library defines the return value of `fread()` to be of the type `size_t`, that is what I have used here.

Once I've read in the data I need to close the file:

```
fclose(f);
```

Now I am ready to process the data in memory using a `for` loop to iterate through all the characters in the array `b` (from `i = 0`, while `i` is less than `size`), incrementing `linecount` whenever a newline character is found:

```
for (i = 0; i < size; i++) {
    if (b[i] == '\n') {
          linecount++;
    }
}
```

Freeing Memory

Right at the end I call `free()` on the buffer `b`. The `free()` function frees up the memory that was allocated with `malloc()`. If I didn't free up that memory, it would remain allocated – that is it would remain unavailable for use by anything else in my program. So even though I finish with my buffer variable, `b`, once I exit the `readin()` function, the memory that I allocated would still be in use, taking up space that I might need for other things later on.

Memory Leaks

But since my local variable `b` is no longer available to me once I've left this function, I no longer have any way of getting at that bit of memory. I no longer have a variable that points to that memory location. This is called a memory leak. In a short and simple program like this, a single memory leak may not cause any problems. But in a large program, if you accumulate memory leaks your program may eventually run out of memory causing it to slow down or even to crash. So, when you allocate memory that you no longer need be sure to free it!

Garbage Collection

> Some programming languages such as Java, C# and Ruby automatically free up unused memory in a process known as 'garbage collection'. A programming language that does garbage collection relieves the programmer from the responsibility of freeing memory explicitly but it also introduces some performance penalties (garbage collection may slow down your programs). The C language is often faster and more efficient than garbage-collected languages – but you must be careful to manage memory yourself and free up memory that is no longer needed.

Saving records to disk

In the final program in this course, I show how to save, load and modify a database of 'records' stored on disk. This is a bigger program than those which I've created previously. It makes use of many of the techniques and functions from earlier chapters so, in addition to showing examples of file-handling, this program should also help you revise many of the important topics described earlier in this course.

Before looking at the code, let me explain what this program, does. It maintains a database of CD records saved into a binary file called *″cd_database.bin″*. It prompts the user to enter single-letter commands to view or modify the data in this file. The commands available are:

a	**a**dd a new record
d	**d**isplay all existing records
m	**m**odify a specific record
n	show the total **n**umber of records
s	**s**ave data from memory to a file on disk
q	quit

02_CDdatabase

The program comprises three source code files: *main.c* which runs a loop to process the one-letter commands until 'q' is entered; *mystring.c* which contains my line-reading function `readln()`, and *cddb.c* which contains the bulk of the code to manipulate the CD database.

There are also two header files: *mystring.h*, which declares the functions in *mystring.c*; and *cddb.h*, which declares the functions in *cddb.c* as well as an `enum`, a string type `Str50`, the `CD struct` type, a globally available array `cd_collection` (actually this is declared a pointer to a `CD` struct `CD *cd_collection` which, in this program will be the address of an array of `CD` structs), and a `CD struct` called `tempcd` which I use as a temporary storage area when I read in the data for a new CD as it is entered at the keyboard.

I won't discuss the entire code of this project. That would take far too long and, in any case, you should by now be able to understand most of it. Instead, I want to concentrate on some functions and techniques which I have not used before.

Let me look first at the `load_cdcollection()` function that loads data from the disk file. This function is called by the `display_cdcollection()` function when the user enters 'd' at the command prompt.

This is the code of the function:

```
static int load_cdcollection(char *filename) {
    FILE *f;
    int numrecs;
    int numrecsread = 0;

    numrecs = number_of_records_in_db(filename);
    f = fopen(filename, "rb");
    if (f == 0) {
          printf( "Cannot read file: %s\n", filename);
    } else {
          cd_collection = realloc(cd_collection, sizeof(CD) * numrecs);
          numrecsread = fread(cd_collection, sizeof(CD), numrecs, f);
          if (numrecsread != numrecs) {
                printf("Error: %d records in file but
                      %d were read into memory", numrecs, numrecsread);
          }
          fclose(f);
    }
    cdarraylen = numrecsread;
    return numrecsread;
}
```

The function is declared as `static` because I want it to be 'private' within this code-file. The code begins by calling another function `number_of_records_in_db()` which returns the number of records stored in the data file. I'll explain how it does this shortly. Then it tries to open the file in binary read ("`rb`") mode. If this operation is unsuccessful the return value assigned to `f` is `NULL` or 0 and I display an error message. Otherwise, `f` is assigned a pointer to the file.

I read in all the data from the file using `fread()`:

```
numrecsread = fread(cd_collection, sizeof(CD), numrecs, f);
```

The first argument, `cd_collection`, is the array into which the records will be read. The second argument is the size of each item – each block of data represents a `CD struct` so I give its size as `sizeof(CD)`. The third argument is the number of items to be read and the fourth argument is the file pointer. The returned value is the number of items that were read in and this is assigned to the `numrecsread` variable.

This gives me a way of testing that the number of items which I tried to read and the number of items that I actually read are the same and, if not (due to some unforeseen error), I display an error message:

```
if (numrecsread != numrecs) {
   printf("Error: %d records in file but %d were read into memory",
         numrecs, numrecsread);
}
```

When I've finished processing the file, I close it:

```
fclose(f);
```

But how did I calculate the number of records that were in the file before I began reading them? Recall that this value was stored in `numrecs` and I used this variable in two places. As mentioned above, I used it with `fread()` to specify the number of records to read into the array `cd_collection`:

```
fread(cd_collection, sizeof(CD), numrecs, f);
```

But *before* I did that I used it in this line of code:

```
cd_collection = realloc(cd_collection, sizeof(CD) * numrecs);
```

In my previous example programs the `cd_collection` array was declared to be of a fixed length. For example, if I wanted to store 4 `struct`s in the array, I declared the array as `cd_collection[4]`. But in this program I don't know how many `struct`s I need to store until I calculate how many records exist in the file on disk. It could be 4 or it could be 400. And I need some way of setting aside enough memory for `cd_collection` to accommodate all the records read in from the file. To do this I use the `realloc()` function. We've used `malloc()` previously to allocate memory for an array. In *01_ReadFile*, I allocated enough memory to store characters read from a file in the char array, `b`, where `size` was the size in bytes or characters of the file:

```
b = (char*)malloc(size);
```

Similarly, in the current project, *02_CDdatabase*, I use the `malloc()` function in `create_cdcollection()`to set aside enough space for 4 `CD struct`s like this:

```
cd_collection = (CD*)malloc(sizeof(CD) * 4);
```

The `realloc()` function reallocates memory that was previously allocated using `malloc()`. This allows me to change the amount of memory allocated. But this still doesn't explain how I calculated the number of records in the data file. This was done by this function:

```
int number_of_records_in_db(char *filename) {
    FILE *f;
    int endpos;
    int numrecs = 0;

    f = fopen(filename, "rb");
    if (f == 0) {
          printf( "Cannot open file: %s\n", filename);
    } else {
          fseek(f , 0 , SEEK_END);
          endpos = ftell(f);
          numrecs = endpos / sizeof(CD);
          fclose(f);
    }
    return numrecs;
}
```

This function moves the file pointer to the end of the file:

```
fseek(f , 0 , SEEK_END);
```

It calls the `ftell()` function to return the position of the file pointer – that is, in effect, the total size of the file, and it assigns this value to the `endpos` variable. In order to calculate the number of records in the file, I just need to divide the size of the file, `endpos`, by the size of a single `CD` record. This tells me how many `CD` records fit into a file of this size. This is how I do that calculation:

```
numrecs = endpos / sizeof(CD);
```

When `CD` records are loaded from a file into the `cd_collection` array, the total number of records is saved in the variable `cdarraylen`. This variable is used later when I save the records from the array into a disk file in the `save_cdcollection()` function. In my program, I save this data into a backup file called "*cd_database.bak*". The data-saving routine is very straightforward. First I open the file in binary write (`"wb"` mode. Then I use `fwrite()` to save data from `cd_collection` with each item having the size of a `CD struct`, and the total number of items given by `cdarraylen`:

```
fwrite(cd_collection, sizeof(CD), cdarraylen, f);
```

The rest of the code in this short function is used to check that the operation has completed correctly, then the file is closed and some messages are displayed:

```
void save_cdcollection(char *filename) {
    FILE *f;
    int count;

    f = fopen(filename, "wb");
    if (f == 0) {
          printf( "Cannot write to file: %s\n", filename);
    } else {
          count = fwrite(cd_collection, sizeof(CD), cdarraylen, f);
          if (count != cdarraylen) {
                printf("initialization failed\n");
          } else {
                printf("saved\n");
          }
          fclose(f);
    }
}
```

To add a new CD to the data file, I call `add_cd()`. This begins by calling my `readcd_data()` function which prompts the user to enter data for each field of a `CD` record, converts strings to integers when appropriate and does a bit of simple error checking. At the end of all that it initializes the fields of a global `CD struct` called `tempcd`. To add the new data stored in `tempcd` to the existing data file, the `add_cd()` function opens the file in binary append mode (`"ab"`) and calls `fwrite()` to write one `CD`'s worth of data into it:

```
void add_cd(char *filename) {
    FILE *f;

    readcd_data();
    f = fopen(filename, "ab");
    if (f == 0) {
          printf( "Cannot write to file: %s\n", filename);
    } else {
          fwrite(&tempcd, sizeof(CD), 1, f);
          fclose(f);
    }
}
```

Notice, by the way, that `fwrite()` requires its first argument to be the address of the data to be written. When I called `fwrite()` previously, that argument was `cd_collection` – which is an array. In C, an array is an address in memory, so that needs no special syntax when used with `fwrite()`. With a `struct`, however, I need to pass the *address* of the `struct` which is why I have had to use the ampersand (`&`) 'address-of' operator: `&tempcd`.

Finally, I want to look at the code needed to find a specific CD record in the file and modify the data which it contains. This begins with the `modify_cd()` function. This prompts the user to enter the number of a CD – that is, the index of the CD in the data file where 0 indicates the first CD. There is a minor problem here.

I need to convert the string entered by the user to a number using the `atoi()` function. That function returns 0 to indicate an error when it is unable to convert the string. But in this case, 0 is a valid number when the user enters it to mean the first record at index 0. My solution is to treat the string "0" as a special case and flag an error by setting an `error` variable to 1 when `atoi()` returns 0:

```
if (input[0] == '0') {
cdnum = 0;
} else {
    cdnum = atoi(input);
    if (cdnum  == 0) {
          error = 1;
    }
}
```

After a bit more error-checking this function calls `change_cd()` – and that is where the real work is done. This is the `change_cd()` function:

```
static void change_cd(char *filename, int cdnum) {
    FILE *f;
    CD* cdptr;
    size_t r;

    f = fopen(filename, "rb+");
    if (f == 0) {
          printf( "Cannot open file: %s\n", filename)
    } else {
          cdptr = (CD*)malloc(sizeof(CD));
          r = fseek(f, cdnum * sizeof(CD), SEEK_SET);
          r = fread(cdptr, sizeof(CD), 1, f);
          readcd_data();
          strcpy(cdptr->name, tempcd.name);
          strcpy(cdptr->artist, tempcd.artist);
          cdptr->trackcount = tempcd.trackcount;
          cdptr->rating = tempcd.rating;
          r = fseek(f, cdnum * sizeof(CD), SEEK_SET);
          r = fwrite(cdptr, sizeof(CD), 1, f);
          fclose(f);
    }
}
```

This opens the file in binary read-and-write mode (`"rb+"`) It allocates enough memory for a single CD record and assigns this to a CD pointer `cdptr`:

```
cdptr = (CD*)malloc(sizeof(CD));
```

It then seeks to a position in the file given by the integer `cdnum`. For example, if `cdnum` were 0 it would seek to the first CD record (at index 0), if `cdnum` were 7 it would seek to the eighth CD record (at index 7). In order to seek to the appropriate position it multiplies `cdnum` by the size of a `CD` record. This moves the file-pointer by the correct number of bytes through the file:

```
fseek(f, cdnum * sizeof(CD), SEEK_SET);
```

The final argument, the constant `SEEK_SET`, specifies that the starting point of the seek operation is the beginning of the file. My code then reads in one CD's worth of data from the current position in the file:

```
r = fread(cdptr, sizeof(CD), 1, f);
```

Now my `readcd_data()` function is called to prompt the user to enter some new data for the CD record. As before, this data is stored in `tempcd`. And finally the data from `tempcd` is copied into the `struct` pointed to by `cdptr`:

```
strcpy(cdptr->name, tempcd.name);
strcpy(cdptr->artist, tempcd.artist);
cdptr->trackcount = tempcd.trackcount;
cdptr->rating = tempcd.rating;
```

Then I seek again to the position of the record which I want to modify and save the new data at that position. As `cdptr` is a pointer variable I can pass this as the first argument to `fwrite()` without having to precede it with the `&` address-of operator:

```
r = fseek(f, cdnum * sizeof(CD), SEEK_SET);
r = fwrite(cdptr, sizeof(CD), 1, f);
```

THE -> OPERATOR

The end result is that I have written new data fields into the record at the specified position in my data file. There is just one more thing here that needs to be explained. What are those `->` operators? It turns out that they are provided for the purpose of accessing the fields of a `struct` to which a pointer is pointing. When you use a `struct` variable such as `tempcd` you can access its fields using a dot, like this:

```
tempcd.name
```

But when you have a *pointer* to a `struct` such as `cdptr` you access the fields using the `->` operator like this:

```
cdptr->name
```

AND FINALLY…

So we finally come to the end of this course. Over the last ten chapters we have covered a lot of ground. If you have followed the course from the beginning you should now be well-equipped to write C programs and understand other people's C programs. There is, of course, much more that you can learn about C – and the most important thing you can do to increase your understanding of C programming is to write lots of C programs. The adventure is only just beginning… where it goes next is up to you!

Appendix

Here are some useful resources for writing and compiling C programs and studying the C programming language.

IDEs/Editors

There are numerous IDEs (integrated development environments) and source code editors that support the C language. In this course, I have predominantly used CodeLite. Here are links to the web sites of some free editors and IDEs for C.

CodeLite

http://codelite.org/

CodeBlocks

http://www.codeblocks.org/

NetBeans

http://www.netbeans.org/

Komodo Edit

http://www.activestate.com/komodo-edit

Web Sites

There are innumerable web sites that contain information and tutorials about the C language. Here are two few sites that I have found particularly useful.

Microsoft C Runtime Library Reference

http://msdn.microsoft.com/en-us/library/59ey50w6.aspx

Tutorialspoint C Standard Library Reference

http://www.tutorialspoint.com/c_standard_library/stdlib_h.htm

Books

Hundreds, maybe thousands, of books have been written about C. If you were to buy just one of these, the book to get is the classic text, "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie. This may not always be a light read but it remains the definitive book on C.

There are also some free online eBooks that you might find useful. "The C Programming Tutorial" is a nice, fairly simple reference to C with plenty of short example snippets of C code:. You can browse online or download it as PDF eBook here: http://www.tutorialspoint.com/cprogramming/index.htm

"The C Cheat Sheet" is another useful, short eBook which is particularly good at pointing out differences between C and Java. So if you have a background in Java, this may come in handy:

http://claymore.engineer.gvsu.edu/~steriana/226/C.CheatSheet.pdf

Bitwise Courses

For more information – and special offers – on my programming courses, be sure to visit the Bitwise Courses web site:

http://www.bitwisecourses.com/