A Beginner's Guide To C Programming V1.3

A guide to the basics of the language

James Bonnyman
Ed. Louise Brown
Louise.Brown@Nottingham.ac.uk

The University of Nottingham
UK

2023
The University of Nottingham

Some common mistakes in C

When starting to learn C, there are a few common mistakes that people make... hopefully having them here will help you as you program (please use the remaining space to add any other ones you think of - and let me know to add them to the list!).

- C is case sensitive - as such, X is not the same as x
- Variables must be defined before they can be used
- New variables do not contain zero when defined
- Each line of code in C ends with a semicolon
- Arrays in C start at zero

Contents

Listings

Chapter 1

Introduction

The aim of this course is to provide you with a solid grounding in the basics of C programming. In addition to this book there is supporting material on the module's moodle page - there is also excellent material available on-line of which I would encourage you to make use.

Some examples that have proved populate with students are

- *https* : *//www.tutorialspoint.com/cprogramming/*
- *https* : *//en.wikibooks.org/wiki/C_Programming*
- *https* : *//www.cprogramming.com/tutorial/c − tutorial.html*

One other link that will be of use is *https* : *//code.visualstudio.com/*. From here you can download a copy of Visual Studio Code, VSCode, the environment we will be using to develop & run code.

Instructions for setting up VSCode for C programming on your own computer are given in Appendix A. VSCode is also available via the Engineering Desktop.

Appendix B provides a guide to starting VSCode, creating and running code.

Your knowledge of C will also be developed through the work in the laboratories associated with the module and during project weeks where you will develop extended code (based on templates supplied) to solve various tasks.

Learning to program will require you to develop skils in the language (getting to grips with the syntax, structure etc.) - the key part however to becoming a good programmer comes from learning to *think like a programmer* , taking time to consider the problem, considering all the stages and then planning your code - the same way an architect would design a building (it is no consequence people who developed large scale systems are called systems architects).

An excellent quote on how **not** to develop code is *"Weeks of programming can make up for hours of planning"!*

Chapter 2

Designing Code

Many people think of coding as simply the writing of the lines that are compiled to produce the program that is executed. While this is clearly part of the task, it is often one of the shortest parts of the development process.

To write code well (and this applies to even the simplest program) there are a number of steps we need to follow is we are to be successful, these are

- Analyse the problem
- Design the solution
- Write the code
- Validate the solution against known data
- Maintain, as required, the code (e.g. following bug reports, OS updates).

A simple case study

In order to look at this in terms of a practical solution, let us imagine that you have been asked to develop a program to solve quadratic equations to obtain values for x1 and x2 using the standard equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

It would be tempting to sit down and write code that performs ONLY the following tasks

- Obtain values for a, b & c
- Calculate X1 and X2
- Display X1 and X2

This however would result in an application that has the potential to fail on many levels!

Let us therefore look at how we should approach this problem.

Analyse the problem

This stage (before the writing of any code) involves talking to people who understand the task for which an application is to be developed, often talking to many different people as each may know parts of the task - only by combining all their knowledge will you completely understand the task in hand.

The information gathered here will often have been given in 'non-technical' terms, it is your job to delve and get the correct level of detail (this can take a very long time).

If we consider this from the viewpoint of an architect being commissioned to design a house, is it equivalent to getting comments such as

- We would like four bedrooms
- A home gym
- An open plan kitchen

These explain what is required, however in very limited detail - the architect would then ask for clarification as required (e.g. would you like a swimming pool in your gym?) in order to be able to draw up the plans required to build the house.

If we go back to our quadratic equation solver, the questions to which you would need answers are

- Will the program need to solve for cases where the solutions are complex
- Are the inputs for a,b & c to be integers or are decimal numbers allowed
- If both roots are the same, should only one be displayed?
- Should the answers be displayed to full precision or to a fixed number of decimal places?
- If a = 0, should the equation be solved (as it is a linear equation) or an error given?

Once we have gathered all this information, we can start the next step - designing the code.

Designing the code

This stage involves taking the information gathered and turning it into specifications that programmers can follow - ideally without any need for further clarification. As such these need to be written with care, specifying exactly what inputs & outputs are required, how tasks are to be completed etc.

It may be that, to aid understanding, designs are presented at a 'high level' with often complex tasks represented as a single 'block' - this 'block' then having its own detailed definition.

Flow charts are still frequently used to show program flow, indeed drawing these can often help highlight areas of additional complexity (and to ensure no 'dead ends').

Going back to our 'house design' analogy... these would be the plans from which the builders, fitters etc. would work form. Different levels of detail may be provided on different plans, for example

- for the builder, it may be sufficient to specify only the location and dimensions of the gym (adding too where doors, windows etc. are positioned) -
- for the electrician, their sets of plans would need to specify where electrical points, lights etc. are to be positioned.
- the gym equipment installers would need specific details on the placement of each item (which should have, if the plans are correct, any power points conveniently placed next to them).

Between the sets of plans there must however be commonality - i.e. the doors must be in the same place on both sets!

In coding we are looking for information to guide the programmer in the development of the code (at this point we may not have even specified the programming language they have to use).

It is worth noting here that we are not telling the programmer(s) **HOW** to write the code, just how the code should function (in the same way we might tell the builder to construct a wall 2m long and 1m high - we do not specify how to build it, just the specifications).

Returning to our quadratic solver, we need to consider the following (which, as we will later see, maps well on to how we write *functional* code.

- What are the inputs (both in terms of number and type)
- What are the outputs (both in terms of number and type)
- What steps are involved

All of which need to be considered with reference to the points raised from the previous analysis of the problem, the 'example' answer to each have been added to the list

- Will the program need to solve for cases where the solutions are complex
  - No - it should indicate such cases cannot be solved and exit
- Are the inputs for a,b & c to be integers or are decimal numbers allowed
  - Decimal numbers should be allowed
- If both roots are the same, should only one be displayed?
  - Yes, with a suitable comment
- Should the answers be displayed to full precision or to a fixed number of decimal places?
  - Display to 3 decimal places
- If a = 0, should the equation be solved (as a linear equation) or an error given?
  - No - a suitable error should be displayed

Whilst the above might seem to provide all the information required, if passed to a programmer they would not be able to complete their tasks! The reason is that for two cases all that is states is *"a suitable error should be displayed"*, this is not specific - the programmer needs to be provided with the actual text to display (this would then allow testing to be undertaken to see if the correct message is displayed).

This would revise the list to be

- Will the program need to solve for cases where the solutions are complex
  - No - it should state "Complex cases cannot be solved" and exit
- Are the inputs for a,b & c to be integers or are decimal numbers allowed
  - Decimal numbers should be allowed
- If both roots are the same, should only one be displayed?
  - Yes - state "Only one root exists, the value for which is " root
- Should the answers be displayed to full precision or to a fixed number of deciamal places?
  - Display to 3 decimal places
- If a = 0, should the equation be solved (as a linear equation) or an error given?
  - No - it should state "Not a quadratic equation (a=0)" and exit

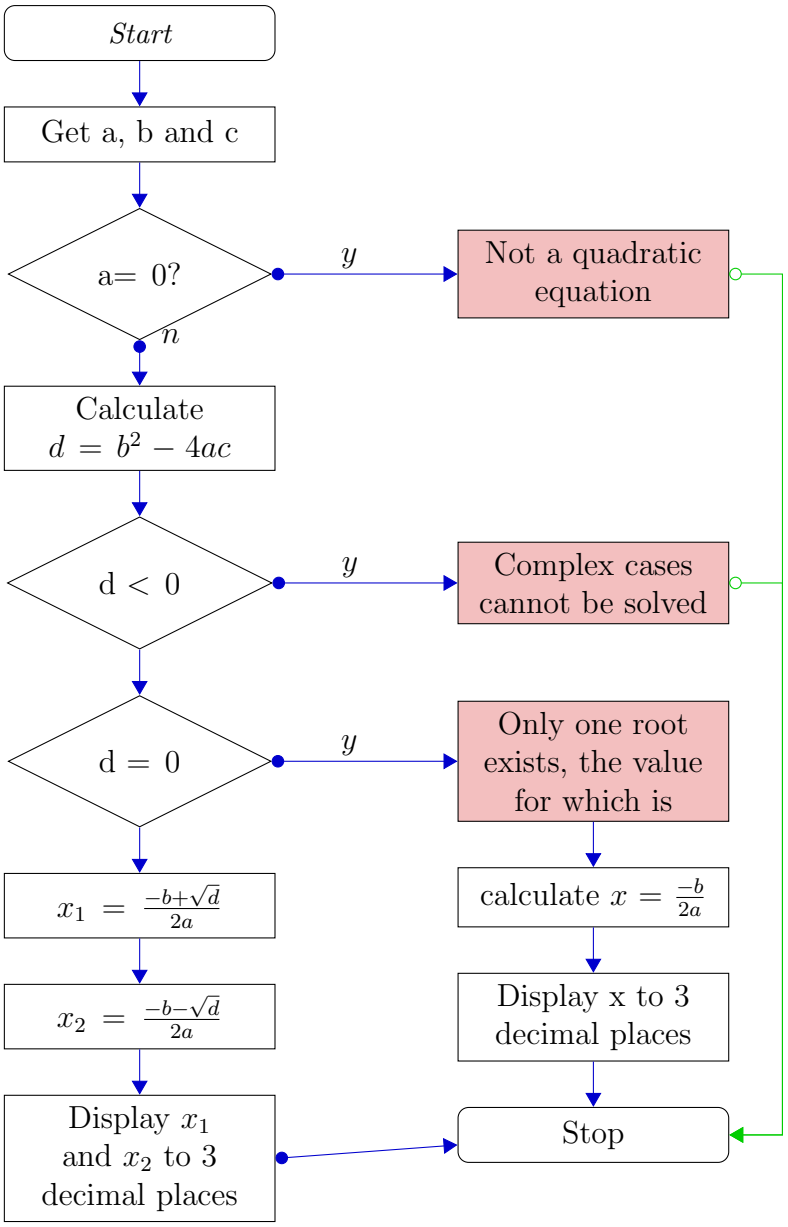


Figure 2.1: Flowchart for solving quadratic equations

As stated before, at this point it is often helpful to draw out a flowchart for the code - in this case it might look something like figure 2.1.

Write the code

Given the flowchart and, ideally some initial testing data, the programmer will develop the code.

Test the code

Once the code has been developed testing begins. This is a critical stage in the development of an application as it allows us to confirm that both individual parts of the application an the application as a whole work as expected.

For this process comprehensive test data must be provided for all possible cases - ideally multiple instances of each (especially where calculations are being performed).

Testing can be both automatic (ideal as this allows banks of test to be repeated following any code changes) however where, for example a graphical interface is used, human testing will also be required.

If we were generating test data for out quadratic solver we might have a table of inputs for a, b & c with the expected output. Some sample test data is provided in table 2.1

a	b	c	output
0	0	0	Not a quadratic equation (a=0)
0	1.1	3.5	Not a quadratic equation (a=0)
1.5	1	4	Complex cases cannot be solved
1	-6	9	Only one root exists, the value for which is x=3.000
2.5	5.0	-2.5	x1=0.414, x2=-2.414
2.5	6	0	x1=0, x2=-2.4

Table 2.1: Sample data for testing.

Maintaining the code

Once code has been developed, tested and validated as working correctly it is tempting to think our work is done, alas this is not the case.

Our programs are dependant on a number of things, including

- The latest operating systems
- The version of the compiler/language
- Security updates release

If any of the above change (e.g. a new version of Windows, IoS is released) we need to retest our application as there may have been changes outside of our control have impacted on the performance of our application.

Ideally we will re-run the tests (this is where automated testing is a huge help) and no problems will be noted. Should however the application not function as originally designed, it would be necessary to identify the problem, make changes as required to the code and repeat the process of validating the application against the original test data.

Chapter 3

Hello World

It is traditional to start every programming course with the program to display "Hello World" on the screen, so here it is (listing 3.1)

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  /* My first program */
5  int main(void)
6  {
7    // This is a single line of comment
8    printf("Hello World");
9    return 0;
10 }
```

Listing 3.1: Hello World Example [c3\hello_world.c]

If we look at this code, there are some key points to note that will help as you learn and develop your own code (note: The formatting italics, bold etc. may be different on the compiler you use - you can often configure this, along with display colours, to your own personal preference)

- Lines of code (e.g. lines 1& 2) starting with a # symbol are actually instructions to the compiler (called preprocessor directives) to modify the code we have written before compiling it - in this case to include text from other files into our code.
- We can add blank lines to make our code more readable (e.g. line 3)
- Every program must have a version of main() - this is 'starting point' of all C programs (line 5). There are different versions of this main(); presented here is the most 'basic' one.
- Blocks of code are contained within { and } (lines 6 and 10 respectively), in this case the block of code belongs to 'main()'
- Commenting code is extremely important (e.g. line 4). This is the *true* C style for a comment - anything between the /* and */ is ignored by the compiler.
- For single lines of comments most compilers (unless we strictly apply the rules of C) allow the style on line 7 - anything following the // is considered a comment
- Lines of code are terminated with a semicolon (e.g. lines 8 & 9)
- The return statement (line 9) within main() ends the program. The value returned from main() is referred to as the exit status; by long standing convention, a value of zero indicates execution completed without any errors.

Code::Blocks (the development environment we use for this module) automatically creates the code for 'Hello World' program for each new project, so why not now create a new project and see this in action (you can find a 'how to' for this in Appendix 1).

As an aside...

As Hello World is usually the first program written when learning a new programming language there is a web site specifically dedicated to it *http : //helloworldcollection.de*

Taken from the site *"Hello World has been implemented in just about every programming language on the planet. This collection includes* ***592 Hello World programs*** *in as many more-or-less well known programming languages, plus* ***human languages****."*

Chapter 4

The very basics of C

In this chapter we will look at some best practice when developing code, moving on to variables and some information about how mathematical operations are carried out (and how to avoid some potential problems).

4.1 Structure and Style when programming

When we write code, we are actually writing a text file which is then compiled and linked to produce the final executable. As a programmer you will, over time, develop your own style for layout (often it is possible to identify who wrote code based on the layout) - sometimes this will be imposed on you (for example by a company's agreed code style).

To the compiler, the layout is somewhat irrelevant - white space and blank lines are (generally) ignored. As such we could write the following lines of code

```
1 #include <stdio.h>
2
3 /* My first program */
4 int main(void)
5 {
6   printf("Hello World");
7   return 0;
8 }
```

Listing 4.1: Well laid out code [c4\well_laid_out_code.c]

As

```
1 #include <stdio.h>
2
3  /* My first program */
4 int main(void) { printf("Hello World"); return 0; }
```

Listing 4.2: Less readable code [C4\poor_layout.c]

While it contains exactly the same code and, as such, will compile and execute with the same output generated it is much harder to read (and debug were there to be an error)

As code gets more complex, line breaks and spaces are even more important. Using an example from the WikiBooks on C, while the following code will compile it is both hard to understand and debug (and does not even fit on a page/screen without a few new line added)

```
1 #include <stdio.h>
2 int main(void) { int revenue = 80; int cost = 50; int roi;
3 roi = (100 * (revenue - cost)) / cost; if (roi >= 0) {
4 printf ("%d\n", roi); } return 0; }
```

Listing 4.3: Difficult to read code [c4\difficult_to_read_code.c]

Written as below (and with the addition of comments that new lines allow) the code is much more readable

```
1  #include <stdio.h>
2  int main(void)
3  {
4    // Declare variables and give initial values
5    int revenue = 80;
6    int cost = 50;
7    int roi;
8
9    // Perform calculation
10   roi = (100 * (revenue - cost)) / cost;
11
12   // Make decision based on value of roi
13   if (roi >= 0)
14   {
15     printf ("%d\n", roi);
16   }
17
18   return 0; // Exit indicating success
19 }
```

Listing 4.4: Reformatted and much easier to read [c4\reformatted_code.c]

Note too the use of indentation of code (line 15) and the use of additional brackets on lines 14 & 16 to make a new block of code which is 'controlled' by the *if* statement on line 13.

This brings us to the (slightly) contentious topic of brackets in C - there are two conventions for this and programmers will never agree on which is correct!

4.2 Brackets and Indenting

If we consider lines 13-16 above, these can be written two ways, firstly as

```
1
2   // Make decision based on value of roi
3   if (roi >= 0)
4   {
5     printf ("%d\n", roi);
6   }
```

Listing 4.5: Aligned bracket style

In this approach the 'open' bracket is placed on a new line and the statement(s) within indented, the closing bracket is placed directly below the opening bracket.

An alternative approach is to have the 'open' bracket is placed on the same line as the 'if', the statement(s) are again indented - the close bracket is,as before, placed on a new line.

```
1   // Make decision based on value of roi
2   if (roi >= 0) {
3     printf ("%d\n", roi);
4   }
```

Listing 4.6: An alternative approach to brackets

Personally (and here Code::Blocks agrees with me!) I use the first approach of having the brackets open/close on new lines. I personally find this easier to work with as you can see the the open/close pairs aligned; you may wish to use the alternative approach (though, on auto-formatting, Code::Blocks will go with my preferred option ◡̈)

As an aside...

There is a contest to see who can write the most obfuscated code (www.ioccc.org).

One example of such code which, when compiled & run, displays the time the code was compiled is

```
main(_){_^448&&main(-~_);putchar(--_%64?32|-~7[__TIME__-_/8%8][">'txiZ^
(~z?"-48]>>";;;====~$::199"[_*2&8|_/64]/(_&2?1:8)%8&1:10);}
```

from: http://www.ioccc.org/years-spoiler.html#2006 , sykes2.c

4.3 Variables

When programming we often need to store values - these can be integers, floats, individual characters or strings (words), these we store in **variables**.

C requires us to declare every variable before it is used (not true for all languages), the term for this is **declaring** - this sets aside the required amount of memory for the type of variable and gives us a 'name' which we use to access (setting or getting) this memory.

In C there are four basic types of variable

- int : Integer values
- char : A one byte variable that can hold a single character
- float : A number that can store a real number (non-integer values)
- double: A variable that can hold real numbers at higher precision than a float

Let us now look at these in detail

4.3.1 int

When we declare a variable of type **int** we are defining one that can hold whole (integer) numbers e.g. 1,2,3,4. The range of values we can store is dependant on the amount of memory the system sets aside for storing an integer & whether we allow positive and negative numbers of only positive ones.

To declare a single integer we use the from

```
int a; // Declare an integer variable a
```

Note that while this declares a new variable, it **does not** default the value to zero - it is up to us to do this. We can do this at the point where we declare the variable

```
int a = 0; // Declare an integer variable a and initialise to zero
```

or split this into two lines

```
int a; // Declare an integer variable a
a = 0; // Set value of a to zero
```

If we wish to declare multiple variables we can do this by first specify the type and then providing a list of variables to declare separated by a comma (if we with to initialise with values we can also do this).

The following example declares the integer variables *Age, YearOfBirth, j*, setting *j* equal to 0 (the other two variables have unknown values until we assign them).

```
int Age, YearOfBirth, j = 0; // Declare multiple variables, set j only to zero
```

4.3.2 char

A char is a variable that is capable of holding a value that represents c standard ASCII character (those you will find on a standard keyboard). It typically takes up one byte however this may not be the case on all systems.

Examples of characters include ‘a’,‘b’,‘c’ - we can also represent ‘special’ characters that we cannot directly type, some of the most commonly used ones are

- ’\n’ - represents a new line
- ’\t’ - represents the TAB key

We declare a **char** variable using the same approach as for an integer, providing the type and then the variables to be declared. As with an integer the value initially stored in the variable is undefined unless we provide an initialiser, e.g.

```
char letter1; // Define a char variable letter1
```

When initialising a char variable there are two approaches - one good, the other bad - we will of course only use the ‘good’ option.

The good way

char letter1 = 'j'; // Declare a char variable letter1 and initialise with the letter j

This approach works as the compiler determines the numerical value associated with the letter 'j' (you can look these up if you google *ascii table*) and stores this in j. It helps us as we can 'read' that letter1 is being initiaised with the letter 'j'

The bad way

char letter1 = 106; // in ASCII, 106 = 'j'

While the end result is the same, from the point of anyone reading the code it is not obvious what character is being stored in letter1 (they would need to refer to an ASCII table).

IMPORTANT

The character '1' (note the single quotes) is NOT the same as the numerical value 1.

If we declared two **char** variables letter1 and letter2 and declared and initialised them as below

char letter1 = '1', letter2 = 1;

The numerical values stored would be

letter1 : 49 (the ASCII value of the charcater 1)

letter2 : 1

Note: In this section we have only covered the storing of individual characters. If we wish to store groups of letters (strings) we use arrays of characters which - this will be covered in section 4.7

4.3.3 float

We use float variables to hold both integer & real (non-integer) numbers. As with other variables, we declare them by specifying the type and then the variable(s) to be declared - remembering that, as with int and char the value is unknown unless we initialise it (or later assign a value).

To identity a float when initialising it, it is good practice to add a 'f' to the value (though the compiler will not complain if you do not!), e.g.

float f = 0.34f, k = 1.233f; // Best practice, an f after the number

float f = 0.34, k = 1.233; // This will also work!

IMPORTANT

It is important to realise that floating point numbers are inexact - this is to say that some values cannot be stored exactly (e.g. 0.1f cannot be represented exactly as a float), this can often lead to problems (especially when performing mathematical operations on a combination of very large & very small values).

This is due to the IEEE 745 Floating Point Representation used to store floats (which we cover in the lectures).

4.3.4 double

As with floats, we use can double variables to hold both integer & real (non-integer) numbers - a **double** simply extends the range of values we can store (noting that a double takes up twice as much memory as a float).

As with other variables, we declare them by specifying the type and then the variable(s) to be declared - remembering that, as with other variables, the value is unknown unless we initialise it (or later assign a value).

Note the initialiser for a double is **l** (rather than d which is what one might expect!).

```
double q = 0.34l, r = 1.233l; // Best practice, an l after the number

double q = 0.34, r = 1.233; // This will also work!
```

4.4 Modifying variable types

When working with integer variable types (**int, char**) we can specify that we need them only to hold positive values (zero in this case being positive), this allows us to double the upper limit compared to that of a signed integer variable (one that can be both positive and negative).

If we assume a system that works on 32 bits, the ranges for an integer a declared as (say) **int a;** are

Minimum: -2147483648
Maximum: 2147483647

If we needed to store a positive integer larger than this we would need to use an alternative variable type (a long integer), this however would take up more memory.

There is however a (potential) solution - if when we declare an integer variable we specify it as being **unsigned int a;** this changes the range as below

Minimum: 0
Maximum: 4294967295

There are other modifiers we can use when programming, these include *static, register, extern, volatile* - a number of these are covered in section 19.

4.5 Mathematical operations

Once we have declared numerical variables we can use them in mathematical operations (adding, subtracting etc.). The basic operations are

- + : Addition
- - : Subtraction
- * : Multiply
- / : Division
- % : Modulus - the remainder after a division (applies to integers only)

> Important: Power Function
>
> One to be careful of - especially if you have programmed in other languages...
>
> In C the ^ symbol does not mean 'to the power of', it is exclusive-or operator (which is covered later in section 6.5).
>
> To calculate $z = x^y$ we use the power function: z = pow(x,y)

For the most part we simply construct equations as we would write them, for example if we had two integer variables a & b (initialised to contain 6 & 7 respectively) which we wished to sum and then place the answer in a third variable c, we could use the following code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5      int a = 6,b = 7,c; // Declare and initialiase as required
6      c = a + b; // Add the values and store in c
7
8  }
```

Listing 4.7: A simple calculation [c4\simple_calculation.c]

Where we need to be especially careful is when mixing variable types or performing calculations where the variable used to store the result may not be appropriate.

Consider the case of doing the simple calculation 1/4 - if we did this on a calculator we would get the answer 0.25.

Looking at this as a programmer we might note that both the 1 & 4 are integers and so declare variables of type **int** for these, we would realise the answer is to be a real number and so would declare a variable of type **float** for the result.

This would give us code as below

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare and initialiase as required
7      int a = 1,b = 4;
8      float ans;
9
10     ans = a / b; // Perform calculation
11
12     // Display answer
13     printf ("\nThe answer is %f",ans);
14
15     // Exit from program
16     return 0;
17 }
```

Listing 4.8: Be careful when mixing variable types [c4\wrong_answer.c]

However, when running the code we would see the following

The answer is 0.000000

which is clearly wrong! The reason for this is that the variables used in the calculation are all integers - as such the calculation will be performed using integers (no decimal parts allowed), this result is then stored in the float variable as 3 (it is automatically converted to 3.0).

This can be a major problem for us (loss of precision, divide by zero problems etc.). We could simply make all variables floats (or doubles) however this would be extremely inefficient (and waste memory).

typecasting variables

There is however a solution, we can temporarily treat a variable as if it were another type (provided a suitable conversion is possible), we do this by a process called **typecasting**.

While this might sound complex, all we actually do it put the variable type we would like, in brackets, before the variable.

So, if we had previously declared an integer variable a as **int a;** and we wished for it to *temporarily* be treated as if it were a float, we would modify if thus **(float)a**.

This is perhaps best seen if we modify the previous code to use typecasting to temporarily treat the two integer variables as if each had been declared as a float

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

int main(void)
{
    // Declare and initialise as required
    int a = 1,b = 4;
    float ans;

    // Perform calculation
    // This time a & b are treated as if they are floats
    // the result of this float calculation is stored in ans
    ans = (float)a / (float)b;

    // Display answer
    printf ("\nThe answer is %f",ans);

}
```

Listing 4.9: Typecasting to avoid problems with calculations [c4\correct_answer.c]

This time when running the code we would see the correct answer

The answer is 0.250000

4.6 Mathematical Precedence

Calculations in C are performed according to the BODMAS convention, namely

Brackets, Operators, Divide, Multiply, Add, Subtract

e.g.

X * Y * Z + A / B - C / D

Can be written (and is calculated as)

(X * Y * Z) + (A / B) - (C / D)

4.7 strings

A string is the term used in programming for a variable that is able to store text (i.e. it is a collection of character which we consider as a single item).

When we declare a string, we need to state the **maximum** number of characters we are going require **plus one extra** to allow for the 'end of string' marker.

In C, to declare a string we do this by

- Specifying **char** as the variable type,
- Provide the name of the variable we wish to declare,
- Append [$\boldsymbol{n}$] to the variable name where $\boldsymbol{n}$ is the number of characters for our string

The snippet of code below shows the creating of some strings of different lengths. As with numerical variables we can declare multiple strings on the same line, we separate the list with commas (we need to specify the size for **each** individual variable.

Note too that, as with numerical variables, a string is not 'empty' on creation.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5    // A variable to store a name (maximum 49 characters)
6    char Name[50];
7
8    // Declare multiple strings on the same line
9    char AddressLine1[100], AddressLine2[100], PostCode[10];
10 }
```

Listing 4.10: Defining a string [c4\declare_a_string.c]

We will see how to read value into strings in section 7.3

Chapter 5

Output

Whilst a lot of code deals with calculations, manipulating data, making decisions etc., users generally need to interact with the program - as such we need to be able to display information and to be able to input information which is then used by our application.

5.1 Displaying information on the screen

In C there are a number of methods we can use to display information on the screen, one of which you have also seen in the small example programs already discussed; this function is **printf**.

> What is a function?
>
> A *function* is a separate block of code that we use to perform a specific task (e.g. calculating the sine of an angle, the area of a circle).
>
> Using functions allows us to break our code into small manageable chunks which can be individually tested and them made use of within our applications.
>
> There are two types of functions
>
> - Standard library functions: Provided as part of the language
> - User defined functions: Ones we ourselves write to perform a specific task.
>
> When using functions we must provide the information required for the task, we may also receive a value back.

printf is a slightly unusual function in that it can take a number of arguments (parameters). At least one parameter is mandatory as the first defines how information will be displayed.

Let us consider a line of code you have already seen

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

If we break this down into its constituent parts

printf() is the standard library function that takes the information we provide inside the parenthesis and performs all the necessary tasks to have this displayed on the screen.

Inside the parenthesis we provide the text to be displayed. As this is a **string** (more than one character) we place the text within double quotation marks. Note: printf requires that the first parameter is a string so, even if you are only displaying one character, all text to be displayed must be within double quotation marks.

The line is then completed with a semicolon - indicating to the compiler that the statement has been completed.

printf, it should be noted, does not add a new line after displaying information, as such is we compile and run the code below

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      printf ("This is my first computer program");
8      printf ("Hello World");
9
10 }
```

Listing 5.1: Outputting to the screen [c5\printf_example_1.c]

Note: When formatting code, LaTeX replaces spaces (within printf statements) with ␣, e.g. "Hello World" becomes "Hello␣World". When you type in code, you need to use a space.

We would see the following output

This is my first computer programHello World

In order to be able to format our text, printf also accepts a number of special formatting options (you may remember them from the section 4.3.2 on char variables), the two most commonly used being

- '\n' - represents a new line
- '\t' - represents the TAB key

If we add these to our code, we can achieve the output we require

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      printf ("This is my first computer program\n");
8      printf ("Hello World\n");
9
10 }
```

Listing 5.2: Outputting to the screen new lines added [c5\printf_example_2.c]

For which we would see the following output

This is my first computer program
Hello World

5.2 Displaying the contents of variables

While we may frequently wish to display fixed text on the screen, there will be many occasions when we wish to display the value stored within a variable.

As the first parameter passed to printf is a fixed string to display, we cannot simply place the variables here as they would be treated as text to display, the example below shows this

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

int main(void)
{

    // Declare some variables
    int a = 1, b = 2;
    float f = 1.23f;

    // Use printf display text on the screen
    printf ("The variables are a, b and f");

    // Exit from main
    return 0;

}
```

Listing 5.3: How NOT to display variables on the screen! [c5\not_displaying.c]

From which the output is

The variables are a, b and f

To get round this problem we use **place holders** into which printf substitutes values as it writes out information to the screen.

Each variable type has its on specific place holder type (there may be more than one - allowing us to display say, an integer, in different format). We can also add additional formatting parameters to modify the output (e.g. specifying to how many decimal places a float or double should be displayed).

A few of the more commonly used ones are (a quick google search on *formatting in c* will allow you to find a more extensive list).

- %d - int (or you can use %i)
- %ld - long int (or you can use %li)
- %f - float
- %lf - double
- %c - char
- %s - string
- %x - hexadecimal

Important

When using place holders with **printf** you must ensure you provide a variable of the correct type for each variable place holder. If you fail to do so, your code will run however it will produce very 'odd' results!

The compiler will warn you of any missing/mis-matched variables with printf - a good reason to always look at warnings!

Going back to our code, to be able to display the contents of the three variables we need to modify the printf formatting string (the text inside the double quotation marks) indicating where

we would like each variable to be displayed. We the provide the variables as parameters to printf, separating items with a comma.

The code below has been updated to show this, new lines have also been added to show how \n can be used within the formatting string.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      int a = 1, b = 2;
8      float f = 1.23f;
9
10     // Use printf display text on the screen
11     printf ("The variables are\na = %d\nb=%d\nf=%f", a, b, f);
12
13     // Exit from main
14     return 0;
15 }
```

Listing 5.4: Displaying contents of variables on the screen! [c5\displaying_variables.c]

which gives the output

The variables are
a = 1
b=2
f=1.230000

Improving the display of numbers (and text)

When displaying numbers on the screen we may wish to limit (in the case of floats or doubles) the number of decimal places we show, for integers we might wish to ensure the same 'width' is used when displaying a value (padding with space if required) to have neat columns of numbers.

To do this this we expand the information in the place-holder (%d, %f etc.) to provide both information on the width (in characters) to use to display the value (note: if this is insufficient, e.g. we indicate two characters to display an integer and the value to be displayed is 1000 the format specifier will be overridden).

Where we specify the width also indicates where any white space will be used, for an integer this would allow us two options

%6d : display integer, 6 characters wide, right justified
%-6d : display integer, 6 characters wide, left justified

For floating point numbers (float, double) we can specify **just** the precision to use (number of deciamal places) or both the width to use and the number of decimal points.

%8.2f : a total width of 8 characters, within the 8 characters the last 2 will hold the decimal part.
%.2f : the minimum width with two decimal points of precision.

Again, as with integers we can left/right justify the output.

The short example following shows the different integer and float formatting statements and the output when the code is run. For east of reading, each **printf** statement is on a its own line (with a newline sent to the screen using \n), we could of course use a single statement with \n

to add new lines on the output.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      int a = 123;
8      float f = 12.456f;
9
10     // Use printf display text on the screen
11     printf ("Examples of integer formatting\n");
12     printf ("a = %d (no modifier)\n", a);
13     printf ("a = %6d (w:6, justify:right)\n", a);
14     printf ("a = %-6d (w:6 )\n", a);
15
16     // Use printf display text on the screen
17     printf ("Examples of float formatting\n");
18     printf ("f = %f (no modifier)\n", f);
19     printf ("f = %6.2f (w:6, 2dp, justify:right)\n", f);
20     printf ("f = %-6.1f (w:6, 1dp)\n", f);
21
22     // Exit from main
23     return 0;
24 }
```

Listing 5.5: Formatting the output of numbers [c5\formatting_numbers.c]

The output from which is

```
Examples of integer formatting
a = 123 (no modifier)
a =    123 (w:6, justify:right)
a = 123    (w:6 )
Examples of float formatting
f = 12.456000 (no modifier)
f =  12.46 (w:6, 2dp, justify:right)
f = 12.5   (w:6, 1dp)
```

Chapter 6

Operators in C

In C we have already seen how we can use operators to perform mathematical operations (add, subtract etc. - see section 4.5). This chapter details the additional operators, e.g. to compare things (which we call relational operations) available to us when programming in C

6.1 Increment and Decrement operators

C also provides additional increment and decrement operators which allow us to add or subtract one from a variable.

- The increment operator is written as ++
- The decrement operator as −−

The position of the increment/decrement operator is also important, especially when the result is being assigned to another variable. Placed **before** the variable results in the increment/decrement being performed **before** the assignment, **after** the variable results in the increment/decrement **after** performed before the assignment, this is perhaps best shown by the example code below

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      int a, b;
8
9      // Note: Order of a,b indicates the order in which operations are carried out on execution
10
11     // Increment operators
12     b = 3;
13     a = ++b; // b is now 4, a is also 4
14     a = b++; // a is 4, b is now 5,
15
16     // Decrement operators (reset a back to 3)
17     b = 3;
18     a = b--; // a is 3, b is now 2
19     a = --b; // b is now 1, a is also 1
20
21     return 0; // Exit from main
22 }
```

Listing 6.1: Increment and Decrement operator example [c6\inc_dec_examples.c]

6.2 Relational operators

Relational operators are used to make comparisons between variables, for which the result must be either *true* or *false*.

The majority of the relational operators will be familiar to you from your studies of mathematics, two however may catch you out as you start to program in C.

Table 6.1 shows the basic relational operators (answers are based on a=10, b=3).

Operator	Description	Example	Result
==	Checks if values are equal	(a == b)	false
!=	Checks if values are not equal	(a != b)	true
>	Checks if the left operand (value) is greater than the right	(a > b)	true
<	Checks if the left operand (value) is less than the right	(a < b)	false
>=	Checks if the left operand (value) is greater than or equal to the right	(a >= b)	true
<=	Checks if the left operand (value) is less than or equal to the right right	(a <= b)	false

Table 6.1: Relational operators in C (for the examples, a=10,b=3)

Warning!!!

One to be careful of - especially if you have programmed in other languages...

To compare two values in C you must use ==

Using a single = will not cause an error - it will however perform an assignment (make one thing equal to another).

This catches many people out (even experienced programmers!)

6.3 Logical operators

Logical operators allow us to make more complex tests by combining the results of individual relational tests, e.g. if we wanted to perform a task if a is 7 or a is 10.

Table 6.2 shows the basic relational operators (answers are based on a=7).

Operator	Description	Example	Result
&&	The is a Logical **AND** operator. If both the operands (values) are true then the condition equates as *true*	(a == 7) && (a==10)	false
||	The is a Logical **OR** operator. If either the operands (values) are true then the condition equates as *true*	(a == 7) || (a==10)	true
!	The is a Logical **NOT** operator. It reverses the logical state of a condition	!(a == 3)	true

Table 6.2: Logical operators in C (for the examples, a=7)

6.4 Assignment operators

C also provides us with a number of **assignment operators**, in some ways you can consider these to be a form of *short hand*.

Table 6.3 shows the assignment operator version and how this would be written *long hand*.

Assignment operator	long hand version
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Table 6.3: Assignment operators in C

As assignment operators work by storing the value to the right of the operand in the location on the left it is possible to *chain* assignments e.g.

```
a = b = c = 0;
```

which would assign zero to all three variables. This is however somewhat difficult to read so you might be better of simply writing

```
a=0;
b=0;
c=0;
```

6.5 bitwise operators

Bitwise operators look at the individual bits that make up an integer number and return the logical result (in the same way as covered in digital electronics).

Assuming A=60 (00111100 in Binary) and B=13 (00001101 in Binary)

Operator	Description	Written as	Bits	Result (binary)	Result (Decimal)
&	Perform a bitwise AND	A & B	00111100 & 00001101	00001100	12
||	Perform a bitwise OR	A || B	00111100 || 00001101	00111101	61
^	Exclusive OR (XOR)	A ^ B	00111100 ^ 00001101	00110001	49
~	Ones compliment 'flips' bits	~A	~00111100	11000011	195
<<	left shift bits 'n' places	A << 2	00111100 << 2	11110000	240
>>	right shift bits 'n' places	A >> 2	00111100 >> 2	00001111	15

Table 6.4: Bitwise operations

Chapter 7

Input: Reading in information

As well as displaying information on the screen, we often need to take input from the keyboard. In C (as with many other programming languages) there are often many ways to perform the same task - this chapter will present the 'typical' approach for each, starting with the function that provides a method for reading any variable type, **scanf**

7.1 scanf - for most things...

scanf is the 'partner' function to printf - it allows us to read in different variable types **provided** we state what the types to be read. scanf uses the same formatting characters as printf so, to specify we will be reading an integer we would need to use %d, for a float %f etc.

When using scanf there is one **very important** thing to remember - we **must** prefix the variable in we will be storing the input with a & (the reasons for this will be covered in chapter 13).

The listing below shows how to read an integer and a float into declared variables, also shown is the code to then display these values on the screen.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int a;  // Declare some variables
7      float f;
8
9      // Use printf to prompt the use to enter an integer
10     printf ("Please enter an integer\n");
11
12     // use scanf with %d to read into 'a'
13     scanf ("%d",&a); // note the important &
14
15     // And display on the screen
16     printf ("The value you entered for a is %d\n", a);
17
18   // Use printf to prompt the use to enter an float
19     printf ("Please enter a float\n");
20
21     // use scanf with %f to read into 'f'
22     scanf ("%f",&f); // note the important &
23
24     // And display on the screen
25     printf ("The value you entered for f is %f\n", f);
26
27     return 0; // Exit from main
28 }
```

Listing 7.1: scanf - for reading values into variables [c7\scanf_example_1.c]

Reading individual characters & strings from the keyboard

We can use scanf to read chars (using %c) & strings (using %s) however there are better functions for this which you are strongly advised to use for these (covered in section 7.3.2)

It is possible to use scanf to read multiple values from the keyboard and to store these in a collection of variables - the problem is that the formatting is strict and if users do not enter the data as specified things can get somewhat *confused*!

If you do wish to read multiple values, the formatting strings specify the list and type of variables to be read - we then add each variable into which the information will be stored into the scanf statement, separating each with a comma - a remembering to include the (all important) & before the variable name.

The code snippet below gives a few examples

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      // Declare some variables
6      int a,b,c;
7      float f,g;
8
9      // Use printf to prompt the use to enter 3 integers
10     printf ("Please enter three integer\n");
11
12     // use scanf with %d to read into a, b and c
13     scanf ("%d %d %d",&a, &b, &c); // note the important &
14
15     // And display on the screen
16     printf ("The values entered were %d %d %d\n", a, b, c);
17
18   // Use printf to prompt the use to enter an float
19     printf ("Please enter two floats\n");
20
21   // use scanf with %f to read into f ang g
22     scanf ("%f %f",&f, &g); // note the important &
23
24   // And display on the screen
25     printf ("The value you entered were %f and %f \n", f,g);
26
27     return 0; // Exit from main
28 }
```

Listing 7.2: Reading multiple values with scanf [c7\scanf_example_2.c]

In the above example we had spaces between the format specifiers (e.g "%f %f"), had we used (say) a comma (e.g. "%f,%f") we would need to inform the user to put commas between their input values.

As users often do not follow instructions it is often easier to ask for values 'one at a time'.

This approach also has the advantage as we can, if required, validate user input one item at a time, e.g. if we wanted three integers each in the range 1-10 we could ensure each one was entered correctly before asking for the next.

7.2 reading individual characters

While it is theoretically possible to use scanf to read an individual character from the keyboard it is not ideal - not least as the user has to press the 'return key' (which is itself a character).

The standard C library function we us in C to read a single character from the keyboard is **getchar()**. As it is a function it requires () however as we do not need to pass any parameters for it to function, nothing should be placed within these. It returns a char which we can then store in a suitably declared variable (most efficiently a char).

To output a char in the screen we have the option of using either printf with the %c formatting string or the function putchar() which outputs a single character to the screen - both are demonstrated in the example code below.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // declare variable
7      char c;
8
9      // Wait for a kepypress, store result in c
10     c = getchar();
11
12     // Display on the screen using printf
13     printf ("The charcter pressed was %c\n", c);
14
15     // Display the charcter using putchar()
16     putchar(c);
17
18     return 0; // Exit from main
19 }
```

Listing 7.3: Capturing and displaying single characters [c7\getchar_example.c]

getch and*putch*

If we have a compiler that supports **conio.h** we can use getch() and putch() as substitutes for the getchar() and putchar() functions.

Note that **conio.h** (and so the functions it provides) are not part of the C standard library or ISO C, nor is it defined by POSIX.

7.3 reading strings

7.3.1 The scanf approach

As with reading a single character, we could make use of scanf with a %s format specifier - and indeed for single words this would work perfectly well, for example to request from the user a person's name and to then store it in the variable *name* we could use the code as below.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      char name[50];
8
9      // Wait for a kepypress, store result in c
10     printf ("Please enter your name");
11
12     // read in using scanf with %s
13     scanf ("%s",name);
14
15     // Display on the screen using printf
16     printf ("Hello %s\n", name);
17
18     // Exit from main
19     return 0;
20 }
```

Listing 7.4: Reading strings with scanf [c7\string_with_scanf.c]

Where is the **all important** & in the scanf?

Note In the above example will have spotted that there is no & before the variable name on the scanf statement - this is based on the fact that a string is a collection of characters and so the compiler can implicity determine the information the & would provide.

The explanation for this is provided in section 17.

This approach works well until we have spaces! When scanf 'sees' a space it assumes the input is finished and dismisses what follows (or uses it as the input for the next variable if we are reading multiple variables using scanf); as such had we entered 'Jo Bloggs' as the name in the above example, only the text 'Jo' would have been stored in name.

As we often need spaces (e.g. in a file path such as "My Documents") we need an alternative, this is provided by gets();

7.3.2 The gets approach

The standard library function gets() allows us to read a single string from the keyboard with all input (including spaces) stored in a previously declared variable. The example below shows how to make use of gets.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5    // Declare variable - maximum 99 characters
6      char str[100];
7
8      // Prompt for text
9      printf("enter some text\n");
10
11     // Store intput in str
12     gets(str);
13
14     // Display a message on the screen
15     printf("you entered : %s\n", str);
16
17     // Exit from main
18     return 0;
19 }
```

Listing 7.5: A better approach to reading strings [c7\string_with_gets.c]

gets - a word of caution...

While *gets()* is very useful as it allows us to read strings containing spaces, it does have a potential 'flaw' which can cause us **major** problems!

If the user enters more characters than we have allocated to the variable it will 'overflow' the variable - which may corrupt other parts of memory (with potentially very *interesting* results.

To write code that stops this happening (a good idea!) we use the function fgets which gets a string from a **specified input stream** rather than gets which reads from the **standard input stream (stdin)**.

When using fgets we therefore need to specify not only the variable but its size plus the input stream to read from - which, when using the keyboard, is **stdio**.

This would change the

```
gets(str);
```

in the above example to become

```
fgets(str, 100, stdin);
```

More details on fgets and other 'file' functions will be covered in section xxx

Note that this is a very subtle approach to reading strings, one which many people would not know to use!

It is presented to help you write the best and most robust possible code from the outset of your programming career.

Chapter 8

Program Flow in Code

When we write code we may need to perform different steps depending on the state of a variable, input from the user etc. This leads us to the concept of program flow (which relates back to the design of a program as covered in chapter 2).

To do this we make use of our relational and logical operations (chaper 6) to create logic based on our requirements, the output of which is used within the decision making capabilities of C.

We refer to statements that determine whether a block of code is executed as **conditional statements**. The most common statement (and this exists in all programming languages), is the **if** statement.

8.1 The if statement

The **if** statement first evaluates the condition, if this equates non-zero (true) it will execute the single statement following it **or** a block of code contained within {}.

When designing code using (using flow charts), we represent an **if** statement as a diamond. The condition is placed inside the diamonds, lines exiting this indicate the possible paths that can be followed and the actions that will result, this is shown in figure 8.1

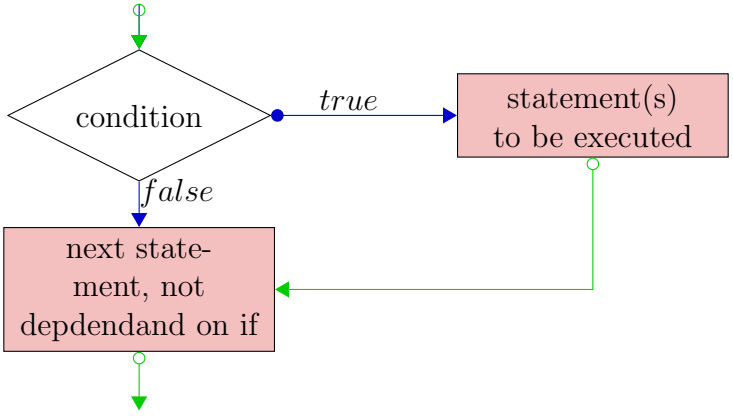


Figure 8.1: Flow for an if statement

When we convert this to code we place the condition within the (). If the condition equates true, the **next line of code** is executed (it helps to indent this line as it adds clarity) - if multiple lines of code are to be dependant on the condition, they **must** be placed within { and } .

Here are a few examples.

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

int main(void)
{
    // Declare some variables
    int a = 7, b=6;

  // A single line of code conditional on the value of a
  if ( a == 7 )
     printf ("The value of a is 7 - so I will do this\n");

  // Multiple lines of code conditional on b not equalling 4
  // so then need to be placed inside { and }
  if ( b != 4 )
  {
     printf ("The value of b is not 4\n");
     printf ("So I will do multiple tasks\n");
  }

    return 0; // Exit from main
}
```

Listing 8.1: An example of if in use[c8\if_example.c]

As C treats a single semicolon as a 'null' line of code, be careful not to place a semicolon after the condition - if you do, this will be the line of code dependant on the condition rather than the line expected.

For this reason you might wish to always use {} to define a block of code controlled by the **if** statement - even though it contains only one line,as per the example below

```c
  // A single line of code conditional on the value of a
  // but within { } to avoid problems with semicolons
  if ( a == 7 )
  {
     printf ("The value of a is 7 - so I will do this\n");
  }
```

Listing 8.2: A safe approach for if conditions

Warning!!

When performing comparisons, remember to use two equal signs!

Using a single equals sign will **assign** a value, not compare against it. If this value is non-zero, the **if** condition will be **true** causing the statements to be executed (two problems for the price of one!).

This is a common mistake when moving to C - the compiler will warn you about the assignment however, as it is permitted in C, it will **NOT** cause compilation to fail.

Checking if a value is within a range

We can combinerelational operators (section6.2) and logical operators (section 6.3) to create more complex tests (which will ultimately equate to true or false).

One common task is to have a condition that is true if a number is within a range (e.g. do something if an integer a is between 1 and 9 inclusive).

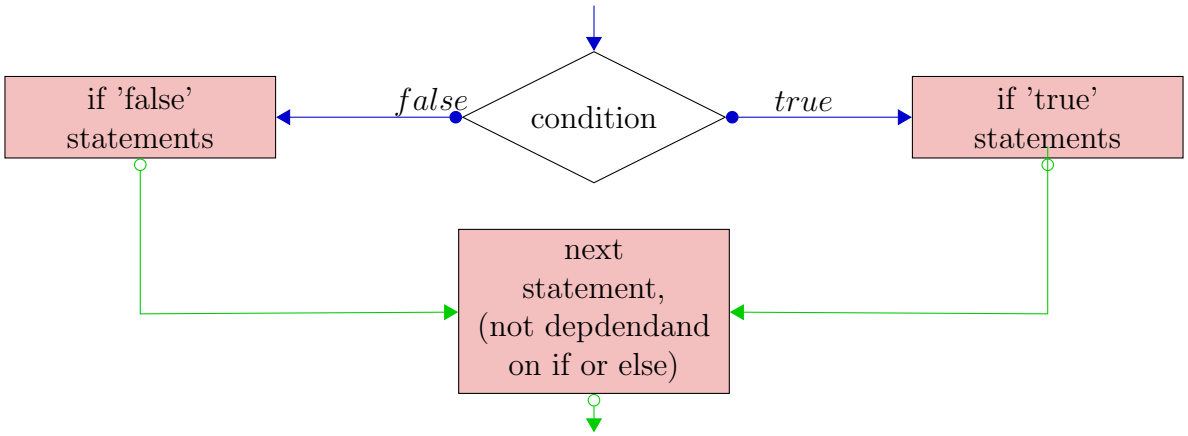


Figure 8.2: Flow for an if/else statement

The condition for this cannot be written (as permitted in some programming languages) as

if (0 < a < 10)

As this statement (for reasons of the order of execution) will always be true!

To write this in C we need to use an approach that tests the upper and lower conditions individually and then determines the overall result based on the individual tests, e.g.

if ((a > 0) && (a < 10)

or, if you prefer

if ((a >= 1) && (a <= 9)

8.2 if/else

The **if** statement allows us to execute code based on a condition being equated as true - we may however wish to have alternative code executed if this is not the case (before moving on to other statements).

To to this we add an **else** option, to which we allocate the code to be executed if the condition equates false (zero).

The else will operate on the **next** statement or on the block of code contained within { } (as before, you may always wish to use the { } approach to avoid problems with semicolons.

Figure 8.2 shows graphically an if/else statement,

Here is a small snipped of code to show how we combine **if** and **else** - { and } are used to show how this can be allied for both cases.

```
1
2   if ( a == 7 )
3   {
4       printf ("The value of a is 7 - so I will do this\n");
5   }
6   else
7   {
8       printf ("The value of a is NOT 7 - so I will do this\n");
9   }
```

Listing 8.3: An example of if/else in use

8.3 if /else if / else if / else ....

We can further extend this to have *if / else if/ else if / else* having as many terms as determined by the requirements of code.

Note that we do not have to use the final **else** - this is just if we need a 'none of the above conditions' were true option.

```
1   if ( a == 7 )
2   {
3       printf ("The value of a is 7 - so I will do this\n");
4   }
5   else if ( a == 8 )
6   {
7       printf ("The value of a is 8 - so I will do this\n");
8   }
9   else
10  {
11      printf ("The value is neither 7 or 8 - I will do this\n");
12  }
```

Listing 8.4: A if/else if/else if .... /else chain

8.4 switch-case statement - a special version of 'if'

If was need to write code where there are a set of tasks to be performed based on an **integer** value, we could use if/else if/else ... (or, if no default case is required, just a series of if statements) to achieve this.

Consider the case for displaying the days of the week based on 0 being Sunday, 1 Monday etc, we could have code as below.

```
1   if ( a == 0)
2   {
3     printf ("Sunday");
4   }
5   else if ( a == 1 )
6   {
7     printf ("Monday");
8   }
9   else if ( a == 2)
10  {
11    printf ("Tuesday");
12  }
13  else
14    //etc....
```

Listing 8.5: One way to write the code is to use if

A large chain of if /else if/ else etc. can be tedious for the programmer and sometimes hard to read, there is however a solution to this which is the *switch-case* construct.

The construct is based on

- The variable to be tested (which Must be of an integer type, i.e. int , char)
- Values against which the variable will be compared
- An **optional** default case

the basic syntax for a switch-case construct is given below

```
1    switch ( /* variable to be compared */)
2    {
3        case /* integer test case */ :
4          /* code to be executed if values match */
5          break ; // end of the lines of code to execute
6
7          case /* next test case */ :
8          /* code to be executed if values match */
9          break ; // end of the lines of code to execute
10
11         default:
12         /* code to be executed if NO values match */
13   }
```

Listing 8.6: General form of a switch-case construct

Please note the colons which are required after the values being compared against - these are easier to see in the sample code below

In operation, the switch-case compares the test variable to the value following the case statement. If the values match the code for the **case** is executed - execution continues until a **break** statement is encountered.

Note that in this, a switch-case differs from the usual syntax of C where a block of code is contained within { and } - in this case it is between the **case** and **break** statements.

We can therefore write our 'days of the week' if / else if ... code using a switch-case construct as below

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      int a = 1;
8      switch ( a )
9      {
10     case 0 :
11         printf ("Sunday\n");
12         break ; // end of the lines of code to execute
13
14     case 1:
15         printf ("Monday\n");
16         break ; // end of the lines of code to execute
17
18     case 2:
19         printf ("Tuesday\n");
20         break ; // end of the lines of code to execute
21
22     // etc...
23
24     default: // If no case is met (OPTIONAL)
25         printf ("\nThe_value_supplied_is_out_of_range\n");
26
27     }
28     return 0;
29 }
```

Listing 8.7: An example of a switch-case construct [c8\switch_example_1.c]

Note: The use of the colons should be easier to see here (as a reminder, they need to be placed after the values of the test cases).

Now consider the case where we might wish the same code to be executed for different values, if were using an **if** statement we could either have repeated statements/code (inefficient) or, better, use a **logical or** to allow for this. e.g. if we wished to perform some lines of code if the variable a was 1, 2 or 3 we would write

```
1   if ( ( a == 1 ) || ( a == 2 ) || ( a == 3 ) )
2   {
3     //code to execute if a is 1, 2 or 3
4   }
```

Listing 8.8: Using or within an if statement

We can achieve the same with a switch-case construct by having multiple cases followed by the code to be executed if any of the cases match, the snippet of code below shows the syntax for this (a case for 3 & 4 added to show further examples).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    // Declare some variables
7    int a = 1;
8    switch ( a )
9    {
10   case 1 :
11   case 2 :
12   case 3 :
13     // Code to do if a is 1, 2 or 3
14   break ; // end of the lines of code to execute
15
16   case 4:
17   case 5:
18     // Code to do if a is 4 or 5
19     break ; // end of the lines of code to execute
20
21   default: // If no case is met (OPTIONAL)
22     // Code to do if no case is met
23   }
24   return 0;
25 }
```

Listing 8.9: The equivalent using a switch-case construct [c8\switch_example_2.c]

A clever trick with the switch-case construct

The switch-case construct work by executing code once a match is found to the point where the **break;** statement is encountered.

If we omit the **break;** statement, all the code for that case **AND** cases following are also executed - stopping only when a **break** statement is encountered.

By way of an example, consider the following example code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare some variables
7      int a = 1;
8      switch ( a )
9      {
10     case 1 :
11         printf ("This_is_case_1\n");
12     case 2 :
13         printf ("This_is_case_2\n");
14     case 3 :
15         printf ("This_is_case_3\n");
16         break ; // end of the lines of code to execute
17
18     default: // If no case is met
19         printf ("This_default_case\n");
20         // Code to do if no case is met
21     }
22     return 0;
23 }
```

Listing 8.10: Omitting the break - it can be useful [c8\switch_example_3.c]

As the value of a is one, the first case statement matches so "This is case 1" will be displayed, the execution continues (as no **break;** has been encountered so "This is case 2" is also displayed - likewise "This is case 3" is also displayed, at which point the switch-case is executed as a **break;** is encountered.

Were a to equal two, the lines "This is case 2" and "This is case 3" would be displayed; were a to equal three only the line "This is case 3" would be displayed.

The default case would be used if a was neither 1, 2 or 3.

Chapter 9

Loops - repeating things

In addition to being able to make decisions on whether code should be executed (using *if* or *switch*) there are often cases when we need to repeat a task a number of times or until a certain condition is met.

It would be impractical to just repeat the lines of code (imagine the time taken if we wished to do something one million times!), such an approach would also be inflexible as it would fix the number of times something will be repeated. The solution is to use a **loop** - of which there are two distinct types

- while loops: which repeat while a condition is true
- for loops: used to count over ranges of numbers

9.1 while loops

This is the most basic form of loop - it continues as long as a condition is true (equates as non-zero). As such (unless we require a loop that never breaks) we must provide a mechanism for the item being tested to change.

By way of a small example, the following code loops until the user enters an age of 0 (zero), note that to 'force' the code into the loop we needed to initially set the test variable **age** to be non-zero (remember: on creating variables can contain **any** value) otherwise the condition might be false and the loop code would be bypassed).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int age = 1;    // Declare variable and initialise to 1
7
8      while ( age != 0) // Loop as long as age is not zero
9      {
10         // Code in {} executed if condition is true (non-zero)
11
12         printf ("\nPlease enter your age");
13         scanf("%d", &age);
14         printf ("You are %d years old\n", age);
15
16         // Code now goes back and repeats the test with the value of age just entered
17     }
18     return 0; // Exit code
19 }
```

Listing 9.1: An example of a while loop [c9\while_loop.c]

9.1.1 Infinite while loops

If, for some reason, we wanted this code **NEVER** to exit we could simply replace the test condition on line 15 with any non-zero value (1 is used by convention) giving

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    int age ;  // Declare variable, no need to initialise this time
7           // as the test condition is not based on its value
8
9      // This is always non-zero, the loop will never break
10     while ( 1 )
11     {
12          // Code in {} executed if condition is true (non-zero)
13
14          printf ("\nPlease enter your age");
15          scanf("%d", &age);
16          printf ("You are %d years old\n", age);
17
18          // Code now goes back and repeats
19          // as the conditions is always non-zero (true)
20     }
21
22     return 0; // Exit code
23 }
```

Listing 9.2: An infinite loop [c9\infinite_while_loop.c]

This type of loop in referred to as an **infinite loop**.

In the above example, to ensure we executed the condition code we needed to provide an initial value for age that was non-zero, in *real code* it is probable this value would have been the result of an input or calculation so, in some circumstances, the loop code may never be executed.

9.2 do-while loops

If we wish to execute statements **before** the test is made we use a variant of the while loop, a do-while loop. In this form the code between the { and } is executed before any test is made.

The following code example shows this approach

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    int age;  // Declare variable, no need to initialise this time
7              // as we read into it before it is tested against
8
9      // The loop is always entered as the test is at the end
10     do
11     {
12         // Code must be executed at least once
13         printf ("\nPlease enter your age");
14         scanf("%d", &age);
15         printf ("You are %d years old\n", age);
16
17         // Test is now made and the code
18         // repeats if the test equates as non-zerp
19         // (i.e. is age is not zero)
20     }
21     while ( age != 0);
22
23     return 0;
24 }
```

Listing 9.3: A do-while loop [c9\do_while_loop.c]

Note: The syntax for a **do-while** statement requires us to have a semi-colon after the test condition. **DO NOT** do this for a while loop (as this then becomes the code controlled by the while statement - most probably not what you intended!).

To summarise the difference

- while
  Test is at the top. Code within { and } executes ONLY if the condition is non-zero (true)
- do-while
  Test is at the end. Code within { and } executes **at least once** and then repeats if the test condition equates non-zero (true)

9.3 for loops

In programming we will often need to count over a range of numbers (e.g. 1 to 100, 90 down to 80), the approach we use to perform this is a **for loop**.

The basic syntax for a **for loop** is

```
1 for ( initialisation ; test ; change )
2 {
3   /* code to be repeated */
4 }
5
6 // next line of code (not part of the loop) would go here
```

Looking at these three parts that make up the for statement (line 1)

initialisation

This is the first thing executed and is generally used to set the start value for the loop (it is possible to omit the initialiser or to use it to declare and initialise variables - this is however out of the scope of this course).

test

Once the initialiasion has been completed the test is made - if it equates true (non-zero) then the loop code is executed; this can be a single line or, if multiple lines are to be controlled by the loop, code within { and } (you may wish always to use the latter approach as it avoids problems with semicolons).

change

After the loop code has been executed the increment (or decrement) is applied - which is (almost invariably) an increment or decrement operation, used to change the value of the loop variable.

Following this the code returns back to the **for** statement and the test is re-evaulated (based on the modified loop variable). If the test equates non-zero (true) the loop is again repeated, the change made etc.

Once the test condition equates zero (false) the code moves on to the next statement following the **for** loop (line 6 in the above example).

Some for loop examples

The following code shows some examples of loops that count up/down, displaying the value of the loop counter as part of the code controlled by the loop.

Note that even in this example, where a single line is controlled by the loop, brackets { and } are used to avoid potential problems with semicolons.

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

int main(void)
{
    // Ddeclare variable and initialise to 1
    int i;

    // Count up from 1 to 100 in steps of 1
    // Note the test could also be: i <= 100
    for ( i = 0 ; i < 101 ; i++ )
    {
        printf ("The value of i is %d\n",i);
    }

    // Count down form 10 to zero
    // Note: we could also use the test: i != 0
    for ( i = 10 ; i >= 0 ; i-- )
    {
        printf ("The value of i is %d\n",i);
    }

    // Count up from 1 to 100 in steps of 3
    // Note the test could also be: i <= 100
    // Increement could also be written as i+=3
    for ( i = 0 ; i < 101 ; i=i+1 )
    {
        printf ("The value of i is %d\n",i);
    }
    return 0;
}
```

Listing 9.4: Examples of for loops [c9\for_loop_examples.c]

Chapter 10

Functions (part 1)

Before we start....

Part one will introduce basic functions - those which (may) take values and return (if required) a single value (e.g. sin, getchar). More complex functions and ways of returning multiple values are considered in chapter 2 - once memory use in C has been discussed.

Up to this point, all the examples presented have code placed within main(). For short programs (even the occasional 'long' one), this is fine however as we look to develop more complex code it leads to many problems (and often increases development time), a few of which are

- Only one person can work on the code at any one time
- Each time we write code, the only way to reuse existing code is via copy & paste
- We cannot validate components in the system (functional testing), just at system level (BAD)

The solution is therefore to break the code up into separate function(s) - blocks of code that 'stand alone' and can be used whenever required - indeed you have already seen how we make use of functions from the standard C library (e.g. getchar, putchar, printf etc.)

When developing a function (as for an entire application) we must consider carefully what it is required to do, what (if any) information it requires to be given to complete its task and what (if any) value it should return.

Once a function has been written we must validate it against testing criteria so we know the function works as expected and, as such, can be used within our application (and perhaps later in other code we develop).

All functions in C consist of three parts, namely

- A return type
- A Name
- An argument list

Plus, of course, some code to do the task!

Let us look at these parts in detail, we can then consider our first function

return type

All functions in C (including main()) are able to return a **single** value which we can then assign to a variable (e.g. x = sin(20); the sin() function returns a float which is then stored in x).

If we do not wish to return a value we must specify this by the use of the the C term **void** (meaning 'nothing').

When starting to define our function we need to be careful to pick the correct return type - which can only be determined if we fully understand the requirements of the function.

name

We are able to give our function **almost** any name we need - it must however meet a few criteria

- We cannot use an existing function name
- The function name MUST start with a letter (upper or lower case) or an underscore

When choosing a name for a function is is helpful to use one that describes what the function does (so helping make code *self documenting*, here are some very bad examples of function names!

- a
- function1
- MyFunction
- DoSomething

Much better is to use an approach where functions are named using first a verb followed by a description of what the function does, here are some good examples based on this approach

- CalulateGainOfCircuit
- DisplayFinalResults
- WriteResultsToFile
- IntialiseLaser

We might even use underscores in the function names to separate parts and so make reading of the names even easier!

the argument list

The argument list is the term used to describe the parameters (variables) we will pass to our function to enable it to perform its defined task (all other information should be contained within the code) - if no parameters are to be passed we must indicate this, again by the use of **void**

The format for the parameters list is a repeated, comma separated list of variable type / variable name pairs of the form

type1 arg1, type2 arg2, type3 arg3, ...

where **type** is any valid C variable type (e.g. int, float, double) and **arg** is the variable we wish to declare

So, if we were looking to define a function that needed to receive three parameters

- x : a float
- y : a float
- l : an integer

The parameter list would be written as

(float x, float y, int l)

Note: The order is not important, you could equally well have

(int l, float x, float y)

The variables we declare here can then be used within the function to enable it to complete its task.

Important: The values we pass to a function are copies of the originals, any changes we make within a function **DO NOT** change the 'original' variables.

The general form

This then leads us to the general form for a function in C

```
type name ( type1 arg1 , type2 arg2, type3 arg3 ... )
{
  /*
    Some code which includes a return of the correct
    type (based on the definition)
  */
}
```

Listing 10.1: General form of a function in C

So from this, applying best practice on function & and variable names here are the definitions for some functions we might create

- float CalculateAreaOfCircle (float Radius)
- int ReturnLargerInteger (int a, int b)
- int CheckIfQuadraticIsComplex (float a, float b, float c)

Now we have considered what a function is, let is look at a simple example to show how we create one that is able to calculate the volume of a cylinder given the radius R and length L, achieved using the function

$$V = \pi R^2 L$$

Before we start we need to consider the inputs (values we need to pass to the functionn) & output (the value it returns) and, **very importantly**, what type of variables to use for each.

For the radius and length it would be sensible to use float as the variable type (to allow for non-integer values). As the calculation will include π this is clearly going to introduce decimal points so the return type will need to support this - as such a float is again sensible.

When selecting a name, we look to use one that describes the purpose of the function - in this case **CalculateVolumneOfCylinder** would seem appropriate.

This then leads us to a function definition of

```
1  float CalculateVolumneOfCylinder ( float R, float L )
2  {
3    /*
4      Some code which includes a return of the correct
5      type (based on the definition)
6    */
7  }
```

Listing 10.2: A simple function performing a calculation

We now need to consider the code to write to enable the function to perform its task - by implementing the equation for calculating the volume in code.

As you start out programming it is often helpful to have a **template** to work to, for functions that are performing a calculation you might wish to adopt the approach below

- Declare, at the start of the function's code a variable **result** of the type the function will return
- at the end of the function have **return (result);**
- Between these, have the required code to calculate the result

This gives the following

```
1  float CalculateVolumneOfCylinder ( float R, float L )
2  {
3    float result;
4    /*
5      code to calculate result goes here
6    */
7    return result;
8  }
```

Listing 10.3: A template for a function

Using this approach means your function is always created in a way that will work (provided of course you implement the calculation correctly!).

For our example, if we add in the code to calculate the volume of a sphere this gives us (note: **M_PI** provides the value for π - we will however need to add include **math.h** at the top of our code (this provides details of mathematical functions and some predefined constants).

```
1  float CalculateVolumneOfCylinder ( float R, float L )
2  {
3    // Declare return variable
4    float result;
5
6    // Calculate value
7    result = M_PI * R * R * L;
8
9    // Return value
10   return result;
11 }
```

Listing 10.4: The completed function

Note that for very simple functions, you can put the calculation directly into the return statement - this is of course your choice!

```
1  float CalculateVolumneOfCylinder ( float R, float L )
2  {
3    // Calculate \& return value
4    return (M_PI * R * R * L);
5
6  }
```

Listing 10.5: Simple calculation done as part of the return statment

Once we have our function, we can then add it to our code and start to make use of it - here is where we need to make a decision.

- Do we put function(s) before any point in the code where they are used?
- Do we wish to put (say) **main()** first and then have our function(s) following
- Perhaps even have the function in another file.

The reason why we need to decide in advance is that we need to 'help' the compiler in its checking of our code - so it canverify we are using our functions correctly.

Placing functions before they are used

If a function is placed at point in the file before it is used the compiler 'remembers' the syntax and can then check we are using it correctly (so helping avoid problems later).

There may be times when this is impossible (for example, two functions make use of each other to complete a task).

Placing functions after the place they are used

In this case, to be able to give the compiler a 'hint' on how the function is to be used (and so enable it to perform checking as it compiles), we place a descriptor for the function at the top of our code, adding a semicolon at the end and omitting the code (we do need to provide it though!).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  // Define HOW the function is to be used, code comes later
6  float CalculateSurfaceAreaOfCylinder ( float R, float L );
7
8  // Function to calculate the volume of a cylinder
9  float CalculateVolumneOfCylinder ( float R, float L )
10 {
11   // Calculate \& return value
12   float Result;
13   Result = (M_PI * R * R * L);
14   return (Result);
15 }
16 int main(void)
17 {
18     // Declare variables
19     float r, l, SurfaceArea, Volume;
20
21     // Obtain values
22     printf("\nPlease enter the radius");
23     scanf("%f", &r);
24
25     printf("\nPlease enter the length");
26     scanf("%f", &l);
27
28     // Get and display the volume
29     Volume = CalculateVolumneOfCylinder(r, l);
30     printf ("\nThe volume is %f", Volume);
31
32     // Get and display the surface area
33     SurfaceArea = CalculateSurfaceAreaOfCylinder(r, l);
34     printf ("\nThe surface area is %f", SurfaceArea);
35
36     return 0;
37 }
38 // Calculate the surface areas of a cylinder
39 float CalculateSurfaceAreaOfCylinder ( float R, float L )
40 {
41   // Calculate \& return value
42   return (2 * M_PI * R * R ) + (2 * M_PI * R * L);
43 }
```

Listing 10.6: Using our own functions (and prototypes) [c10\function_example.c]

Once we have written a function we can call it as many times as we like, from wherever in our code we need (even from within other functions).

10.1 void functions

There are times when coding a function that we do not need a return value, in such cases we need to specify **void** as the return type - we do need a return statement within our function, this time however we just have **return;**

The example below provides an example of this (it is the day of the week printer), the function is below main() to show again how to use a prototype.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Define HOW the function is to be used, code comes later
5  void DisplayDayOfTheWeek ( int Day );
6
7  int main(void)  // Execution starts here
8  {
9      int d; // Declare variable
10
11     // Obtain values
12     printf("\nPlease_enter_a_number_betwwen_0_and_6");
13     scanf("%d", &d);
14
15     // Use a function to display the day of the week
16     DisplayDayOfTheWeek(d);
17
18     return 0;
19 }
20
21 // Function to display day of week - nothing is returned
22 void DisplayDayOfTheWeek ( int Day )
23 {
24     // Display date based on value
25     // Case values on one line as easier to cut/paste :-)
26     switch (Day)
27     {
28       case 0 : printf ("Sunday") ; break;
29       case 1 : printf ("Monday") ; break;
30       case 2 : printf ("Tuesday") ; break;
31
32         /* etc. for other days of the week */
33
34       default:
35           printf ("Invaid_day_provided");
36     }
37     return; // No value needed as the return type is void
38 }
```

Listing 10.7: Example of a void function [c10\void_function_example.c]

Chapter 11

Arrays

In previous chapters we have considered defining individual variables, assigning values to them (either directly, via scanf or from a value calculated by a function) and perhaps then displaying them on the screen.

How though would we manage to store, say, 10000 values? We could declare an individual variable for each however this would be enormously time consuming - if we then needed to pass these values to a function the whole process would break down!

11.1 Defining an array

The solution to this is to declare an **array** - this is allows us to declare a variable that has an *index* that allows us to select a specific item (much like picking a row number in excel); if you have studied matrices then you can picture an array as a matrix.

The approach is much the same as for any variable - you specify the type, the name for the variable - the addition is to add square brackets within which you specify the size of the array.

For example to declare an array of integers, the name of which is Ages and which is to contain ten (or at most ten) values is

```
int Ages[10];
```

In fact we have already seen this - a string you will remember is an array of characters!

When creating this array, as with any other variable the contents is unknown until we populate it; we can do this at the time of declaring the variable by providing a list of values to be placed in the array as below

```
int Ages[10]={12, 15, 23, 11, 19, 6, 5, 1, 2, 3};
```

C also allows us to omit the size, it can work it out for us based on the number of items provided - so we could write the previous statement as

```
int Ages[]={12, 15, 23, 11, 19, 6, 5, 1, 2, 3};
```

A little trick...

Providing just one item when initialising an array will set the remaining values to be zero which can be really helpful when creating large arrays! If we have

```
int Ages[2000]={12};
```

It will declare an array of size 2000, set the 1st value to 12 and the remaining to zero

```
int Ages[2000]={0};
```

Will set the entire array to zero :-)

Multi-dimensional arrays

Arrays do not need only be one dimensional, they can be 2D, 3D - the requirements of the code will define this. The approach for defining them however remains the same, we use square brackets to indicate the size in each dimension, e.g.

```
int Points2D[10][2];
int Points3D[5][10][2];
```

11.2 Accessing items in our array

Very Important

In C arrays start at **ZERO** so an array declared as

```
int X[10]
```

Has elements

X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7], X[8], X[9]

It does not have **X[10]** as this would be the 11th item and we declared it as size 10

Avoid this mistake - it can lead to very serious consequences!

To access values in an array we specify the index to use (remembering to start from zero), you can think of this as the *coordinate* - placing the required index inside square brackets (or sets of for 2D, 3D arrays etc.).

To set the 3rd value (remembering to count from zero) , we would use

```
Ages[2] = 12;
```

To retrieve the value from the 5th item (index 4) and store this in an existing variable A, we would use

```
A = Age[4]
```

To access items in a 2D array we would use the form

```
H = Coordinates[5][2];
Coordinates[8][3] = 11;
```

11.3 string: arrays of characters

You will remember from section 11.3 that we declared a string as an array of char variables, remembering to allocate adequate space for the text and an additional (though never displayed) end of string character.

It is possible to declare strings and initialise them with text (we can also, through use of various string functions, copy text into a string).

To achieve this we can either specify character by character (much the same as specifying individual numbers when declare a (say) integer array), to do this we would use the following approach

As with previous example, we can omit the value in the [] as the compiler can determine this based on the characters supplied.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Main : Execution starts here...
5  int main(void)
6  {
7    // Declare variable - pre-populate the array
8    char msg[] = {'H','i',' ','W','o','r','l','d','\0'};
9
10   // Output with printf
11   printf ("The text is: %s\n", msg);
12
13   puts(msg); // We could also use puts to display the string
14
15   return 0; // Exit the application
16 }
```

Listing 11.1: Initialise string character by character [c11\initialise_string_example_1.c]

Note we have to provide the \0 character - without this, printf (or puts) would not know when the string ended and we could get some very strange output on the screen!

As this would soon get very tedious (and is very hard to read), C allows us to specify the whole string we wish to initialise with, as it is a collection of characters we need to place it within double quotation marks, the example below shows how to do this.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) // Main : Execution starts here...
5  {
6    // Declare variable - pre-populate the array
7    char msg[] = "Hello World";
8
9    // Output with printf
10   printf ("The text is: %s\n", msg);
11
12   puts(msg); // We could also use puts to display the string
13
14   return 0; // Exit the application
15 }
```

Listing 11.2: Initialise string with a string [c11\initialise_string_example_2.c]

You will note this time we not need to add the \0 - this is automatically added at the end for us (as the compiler notes we are initialising with a string, rather than a sequence of individual characters).

11.4 Loops with arrays

Most often when working with arrays we will also be using loops, using the loop variable as the index for our array (remembering that the loop must count from ZERO)

This is best shown via an example - here we declare and pre-populate an one-dimensional array of size 10 with values, We then use a loop to work through array, using printf to display the numbers on the screen.

Of particular importance if the definition of the **for** loop - note it starts at zero and the test condition which when zero (false) will cause the loop to end is

i < 10

Which means the maximum value of i will be 9 (the upper bound of the array), note that we could also use the condition *i <= 9* which would have the same outcome.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) // Main : Execution starts here...
5  {
6      // Declare variables - pre-populate the array
7      int Ages[10] = {12,34,23,11,8,19,6,44,9,16};
8      int i;
9
10     // Loop from 0 to 9 inclusive
11     for ( i = 0 ; i < 10 ; i++ )
12         printf ("Item %d contains value %d\n",i ,Ages[i]);
13
14     // Exit the application
15     return 0;
16 }
```

Listing 11.3: Creating, populating and looping through an array [c11\array_loop_example_1.c]

11.5 Array bounds - be very careful!

When defining an array in C we must be careful not to read/write beyond the size we have defined the as this **cannot be checked by the compiler!**.

Going beyond the 'end' of an array will corrupt memory assigned to other variable etc. with potentially very serious consequences!

This happens as the program, when running, works out where in memory the requested item will reside (based on the base address of the array, the size of each item in bytes and the index of the item to be access) - it then sets/gets the value at that location.

Consider the following declaration of an array in integers followed by three individual integers

```
1  int SampleArray[10];
2  int a;
```

Listing 11.4: Be careful to stay within the bounds of an array

When memory is allocated for these, the memory location for **a** will follow on from the memory allocate for the integer array (the last item being SampleArray[9]). Note: If we declare multiple variables after the array the compiler may put these into memory in a different order to optimise memory usage - which only further complicates the problem!

If we try and access SampleArray[10] we are actually accessing the variable **a**. If we set SampleArray[10] to a value we are changing the contents of **a** which can then cause problems, this can be seen in the example below (the output from running the code is shown below the listing).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Main () - execution starts here
5  int main (void)
6  {
7      // Declare variables
8      int SampleData[10];
9      int a=0;
10
11     // Show initial value of a
12     printf ("Values in a is %d \n", a);
13
14     // Change an array item that IS NOT DEFINED
15     SampleData[10] = 20;
16
17
18     // See how thi affect the value in a
19     printf ("Values in a is %d \n", a);
20
21     // Note: we can retieive this 'invalid' value - it is however 'a' we are getting
22     printf ("Values in SampleData[10] is %d \n", SampleData[10]);
23
24     return (0); // Exit indicating sucess
25 }
```

Listing 11.5: Example of going beyond the end of an array [c11\array_loop_example_2.c]

Output from the code

Values in a is 0
Values in a is 20
Values in SampleData[10] is 20

The problem is that the compiler **cannot** check this for us - it is up to us to use arrays correctly.

Note that this is true both for arrays declared in this method (of fixed size) and when we declare arrays dynamically (covered in chapter 16).

Array Bounds : Take Note!

This does not simply apply to arrays of numerical types (int, float, double) it is equally a problem when defining strings (arrays of char variables) - indeed any type of array.

Be careful to allocate adequate space for a string as often, in order to 'enhance' the user experience, messages to be displayed are revised (perhaps after user testing) and the 'new' message exceeds the length originally allocated.

This results in the same problem detailed above - the characters in our string start to overwrite other variables - causing crashing and/or very unpredictable results.

If is also (for the same reason), when requesting string input from users, to use the alternative method detailed in section 7.3.2 as this can protect code by stopping users entering more characters than the string is declared to hold.

Overwriting arrays is a very common mistake - and one to avoid!

Chapter 12

Variables - Part 2

In the previous chapters we have first at declaring variables within functions (remembering that **main()** is itself a function), later declaring these when we defined functions to provide a method to pass information into functions (to enable them to perform the task for which they were written).

When developing code we need to be aware of the **scope** of a variable and, in particular how this can impact of memory use during the execution of a program.

12.1 Automatic variables

When we declare a variable, space is reserved in memory for this variable, when we assgin a value to this variable (e.g. a=1) we are copying the value into memory, requesting the variable fetches it back from the memory address (location) in which it is stored.

We do not need to know the memory address of the variable (though we can find it as will be seen in chapter 13) - it is handled for us by the operating system.

Such variables declared this way are referred to as **automatic variables** in that they are automatically created for us and removed from memory when the function in which they were declared exits (via a **return** statement).

As such variables declared in **main()** exist for as long as the program is executing, others declared for use in functions will exist only as long as the function in which they are declared is being used - this means that the amount of memory a program is using will change as the code executes.

Note: Although a function may have lines of code which declare variables, these are NOT declared until the function is called.

12.2 Scope of variables

When we write code, we can declare and then use the same variable name in many different functions - indeed it we could not it would be almost impossible to write code!

This leads to the question - how can I have two variables of the same name but have them contain different values? The answer is based on the concept of the scope of a variable.

When we declare a variable it is **private** to the function in which it is declared (there are things called global variables which are to be avoided!), as such there is no *clash* of declarations.

Consider the example below

```
1  // Simple function, counts up to the value passed to limit
2  void SampleFunction1 ( int x )
3  {
4    int i,j,k;
5    return;
6  }
7
8  void SampleFunction2 ( int x )
9  {
10   int i,j,k;
11   SampleFunction1(x);
12   return;
13 }
14
15 // Main : Execution starts here...
16 int main(void)
17 {
18   int i = 1;
19   SampleFunction2(i);
20   return 0;
21 }
```

Listing 12.1: Scope of variables [c12\scope_of_variables.c]

Let us go though this in the order the lines will execute

Line	What is happening
16	Code starts here
19	Variable **i** is declared in main and assigned the value 1
19	SampleFunction 2 is called (passed the current value in i)
8	Local variable **x** of SampleFunction2 is declared
8	Current value in **i** of main copies into **x** (so x=1)
10	Local variables **i, j & k** declared for use by SampleFunction2
11	SampleFunction1 is called, passed the value in x
2	Local variable **x** of SampleFunction1 is declared
2	Current value in **x** of SampleFunciton1 is copied into **x** of SampleFunction2(so x=1)
4	Local variables **i, j & k** declared for use by SampleFunction2
5	Memory for variables declared in SampleFunction1 is released
5	SampleFunction1 exits, code returns to line 15
12	Memory for variables declared in SampleFunction2 is released
12	SampleFunction2 exits, code return to line 23
20	Memory for variables declared in main()is released
20	Program terminates

Table 12.1: Stepping through the example

In this example the **scope** of each variable (where it can be accessed) is **only** the function in which it was declared. We can pass values between functions as parameters and, if a function returns a value, assign the returned value to other variables. We can also reuse variables names in different functions as each variable is declared **locally** and, as such, can be considered **private** to the function.

With this approach, whilst the amount of memory used to store variables throughout the execution of the code will change, at the point where the program exits all the memory that has been allocated automatically has been released - this is how well written code should function (sadly this is not always the case!).

12.3 global variables

It is possible to declare variables whose scope it the entire program code - this is done by creating them outside of any function.

There are many reasons not to use them, for example

- Code can become unstable (and crashes!)
- Functions that depend on global variables cannot easily be reused
- They cam be changed **anywhere** making it very difficult to determine how/when they change

There are a few, very specific, cases when global variables need to be used e.g. interrupt programming, otherwise **do not use global variables!**.

Since they are bad practice, no examples of them will be provided!

Chapter 13

Pointers (part 1)

Before we start....

This chapter looks to **introduce** pointers and show their use with single variables. Pointers can be also be used to access arrays (even to allocate memory for arrays) - this will be covered in Pointers (part 2).

The ability to use memory efficiently is one that distinguishes a good programmer - and can lead to very stable and efficient code.

So far, we have assumed that memory is always available (yet realising that the amount in use can change during a program's execution) - for example, we have assumed that if we declare an array we assume that sufficient space exists in memory for this (not always the case).

We have also noted a that, when using functions we can only return one value - this is not always ideal. As an example, consider solving a quadratic equation - we know that there are two possible solutions so it would be sensible to have one function that could 'return' both answers - possibly also indicating if there solutions could be found.

These problems we can solve through an approach where we start to access the locations where variables are stored, a technique that involves a concept called **pointers**.

Pointers can be one of the most challenging aspects of C to get to grips with, their power is however what makes C such a useful language!

13.1 Pointers

A pointer is a variable type that holds a **memory address** at which is a variable we can access (as long as we are allowed to, for the case of this we will assume we can!).

When working with pointers there are four key things we need to be able to do

- Declare a pointer
- Assign a valid memory address to the pointer (or NULL)
- Access the value at the memory address stored
- Appreciate the relationship between arrays and pointers

Declaring a pointer

Each variable type in C (including types we define ourselves) has an associated pointer type.

The format for declaring a pointer is almost the same as for a 'standard' variable, we simply prefix the name with an asterisk.

The code below shows examples of valid pointer definitions

```
1  int main (void )
2  {
3    // integer pointers
4    int *ptrA, *B, *Data;
5
6    // Float pointers
7    float *pf, *q, *Zvalue;
8  }
```

Listing 13.1: Creating pointer variables

Assigning a memory address to a pointer

Once declared, we will want to assign memory addresses to our pointers - there may be cases however where we will to give them the *equivalent* of a zero value, for this we use the C constant **NULL**.

To have declared all the previous examples and set them to NULL the code would be written as

```
1  int main (void )
2  {
3    // integer pointers
4    int *ptrA=NULL, *B=NULL, *Data=NULL;
5
6    // Float pointers
7    float *pf=NULL, *q=NULL, *Zvalue=NULL;
8
9    // We could also do this on separate lines e.g.
10   int *Another;
11   Another = NULL;
12
13   return 0; // Exit
14 }
```

Listing 13.2: Creating pointer variables and set to NULL

In reality, we will ultimately wish to assign a valid memory address to a pointer. If we have an existing variable we can obtain this through the use of the *address operator* &.

Placing & before a variable will provide the memory address at which the variable resides (specifically, it provides the base address as most variable types span a number of sequential memory address to store a value).

When assigning a memory address from an existing variable to a pointer, the type of variable must match (i.e. an integer pointer takes the memory address of an integer, a float pointer the memory address of a float etc.).

Here are a few examples of creating variables, pointers and then assigning addresses.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (void )
5  {
6      // Integer variables
7      int a, ValueB, d;
8
9    // integer pointers
10   int *ptrA=&a, *B=&ValueB, *Data=&d;
11
12   // Float variables
13   float f,y,z;
14
15   // Float pointers
16   float *pf=&f , *q=&y, *Zvalue=&z;
17
18   // We could also do this on separate lines e.g.
19   int SomeData;
20   int *Another;
21   Another = &SomeData;
22
23   return 0;
24 }
```

Listing 13.3: Assigning addresses of existing variables to pointers [c13\assigning_pointers.c]

13.2 Accessing values via pointers

Once we have assigned a pointer to the address at which a value is stored (i.e. in a previously declared variable), we can access that value (the formal term for this is *pointer dereferencing*.

To do this we make of the * operator, placing this before a **pointer** allows us to get or set the value at that memory location.

The process is perhaps best explained through the use of a small example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (void )
5  {
6      // Declare a in integer
7      int c,d;
8
9    // Declar and integer pointer
10   int *ptrC;
11
12   // Some assgnments
13     c = 10; // C now contains the value 10
14     ptrC = &c; // ptrC now 'Points' to c
15
16     // Get the value of c via the pointer and store in d
17     d = *ptrC; // d now contains 10
18     printf ("\nThe_value_in_d_is_%d", d);
19
20     // Change the value of c via the pointer ptrC
21     *ptrC = 1; //c now contains 1
22     printf ("\nThe_value_in_c_is_%d", c);
23
24   return 0; // exit
25 }
```

Listing 13.4: Accessing variables via pointers [c13\accessing_via_pointers.c]

At first this just looks like a very complicated way of implementing **d=c** - and indeed it is, so the question then becomes... *Why do this?*.

The answer, for single variables, is that it allows us to pass the address of a variable to a function, this function (having taken a copy of this address) is then able to indirectly modify the value.

You have already used this approach, when working with **scanf** - the memory address of variable(s) in which we wish to store the results of **scanf** are provided as parameters, by putting & before the variable name we pass across the address of the variable - not its contents.

Since we can pass as many parameters as we need to a function, we can now (effectively) return as many values as we wish.

So let us now look at functions using pointers!

Chapter 14

Functions (part 2)

Now we know how to get the address of an existing variable, we can look to pass these to a function, providing a method for changing multiple variables with a single function.

To provide a practical example, we will look to develop a function to solve quadratic equations that is able to return back to the caller both the two solutions and (as we are looking to write robust code) a status value which can be used to determine if the calculation could be performed.

To do this we first need to consider the inputs out function will require, based on our knowledge of equation to solve a quadratic equation these will be a,c & c. As we would like a general function we will make these float variables.

The calculated values of x1 and x2 are therefore also going to be floats (reinforced by the fact square roots & division operations are used in the calculation). We will not however consider complex solutions. As we are returning these value via the *parameter list* we will need to declare them as pointers (and when calling the function, remember to pass the address of variables - just as we do with **scanf**).

We also know there are a few cases where we cannot solve the quadratic, these are

- Where a $= 0$: it is not a quadratic equation)
- Where $(b^2 - 4ac) < 0$: the solutions are complex

In such cases we would like our function to return -1 (a is 0) or -2 (complex roots), for a successful calculation we will return 0

This will then lead us to a function based on

```
1 int SolveQuadraticEquation(float a, float b, float c, float *x1, float *x2)
2 {
3   /* Code will go here */
4 }
```

Listing 14.1: Accessing variables via pointers

Error values

If we first consider the error conditions, these we can implement through simple **if** statements, i.e.

- if a is equal to 0, the function is to return -1
- if $b^2 - 4ac < 0$, the function is to return -2

To be efficient, we use an variable into which we calculate $b^2 - 4ac$ and test against this, if all is OK we can then use this when determining x1 & x2 (to avoid an additional calculation).

Calculating x1 and x2

Assuming the tests were passed, we can now calculate x1 and x2. As we have declared x1 & x2 as pointers (remember, in the calling code we need to provide the addresses of existing variables), we prefix these with an asterisk - this then places the calculated values in the relevant memory address, ready to be retrieved when required.

The final function

This then gives us our final working function

```
1  int SolveQuadraticEquation(float a, float b, float c, float *x1, float *x2)
2  {
3      float d; // For storing b^2 - 4*a*c
4
5      if ( a == 0) // Not a quadratie
6      {
7          return -1;
8      }
9
10     // calculate and store b*b-4*a*c for testing ans use later (if OK)
11     d = b*b - 4*a*c;
12
13     if ( d < 0 )
14     {
15         return -1; // Complex
16     }
17
18     // If we have got to here, we can calculate x1 and x2
19     *x1 = ( -b + sqrt (d)) / (2 * a); // Note the use of the * before x1 & x2
20     *x2 = ( -b - sqrt (d)) / (2 * a); // to write to the relevant memory locations
21
22     // As we got here OK, return 0 to indicate all is OK
23     return 0;
24 }
```

Listing 14.2: Quadratic Solver [c14\quadratic_solver_function.c]

All that now remains is to use this from other code (in which we must have declared the variables into which the answers will be stored), the inputs we can pass as absolute values (e.g. 1,2,3) or from other variables.

When calling the function we examine the return value. If the value is zero we are able to use/display the values in x1 & x2, for a return value of -1 or -2 we display a suitable error message.

The example provided uses **if** for this, it could equally well be achieved using a switch-case construct.

Combining this with our function gives a working program as below. Note: **math.h** is required to allow use of the square root function **sqrt**.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int SolveQuadraticEquation(float a, float b, float c, float *x1, float *x2)
6  {
7      float d; // For storing b^2 - 4*a*c
8
9      if ( a == 0) // Not a quadratie
10     {
11         return -1;
12     }
13
14     // calculate and store b*b-4*a*c for testing ans use later (if OK)
15     d = b*b - 4*a*c;
16
17     if ( d < 0 )
18     {
19         return -1; // Complex
20     }
21
22     // If we have got to here, we can calculate x1 and x2
23     *x1 = ( -b + sqrt (d)) / (2 * a); // Note the use of the * before x1 & x2
24     *x2 = ( -b - sqrt (d)) / (2 * a); // to write to the relevant memory locations
25
26     // As we got here OK, return 0 to indicate all is OK
27     return 0;
28 }
29
30 int main (void )
31 {
32     float A,B,C,x1,x2;
33     int retval;
34
35     printf ("Please enter coefficients A,B and C separated by a space\n");
36     scanf ("%f %f %f", &A, &B, &C);
37
38     // Make use of the function
39     retval = SolveQuadraticEquation(A, B, C, &x1, &x2);
40
41     // Use the retval to determine if we can display the answers or an derror message
42     if ( retval == -1 )
43     {
44         printf ("Not a quadratic\n");
45     }
46     else if (retval == -1)
47     {
48         printf ("The solution is complex - I cannot solve these\n");
49     }
50     else
51     {
52         printf("\nThe solutions are x1=%f, x2=%f", x1, x2);
53     }
54     return 0; // exit
55 }
```

Listing 14.3: Code to use our Quadraic Solver Function [c14\quadratic_solver.c]

Chapter 15

Pointers (part 2)

The use of pointers with arrays can be confusing however as, we think of a pointer as something that refers to a single variable - yet an array is a collection of variables (of a type defined in the declaration).

As such, initially the link between a pointer and an array can be a slightly (very) confusing one.

There are two things we need to know

- When we declare an array (e.g. int Data[10]) the variable name points to the 1st item
- A pointer can be indexed like an array

In order to understand what these two statements mean, we need to understand what happens when we declare an array in C.

What happens when we declare an array?

When any variable is declared in C, memory is set aside for the storage of this variable (the actual amount of memory being based on the variable type) - this memory is then reserved until the function exits.

When we declare an array, the same process is applied however rather than a single memory location being allocated, a **continuous** block of memory is reserved which is the exact size required (based on a calculation of $array_size * size_of_variable$).

Each item in the array has its own memory address (which we can access put placing an & before the index item, e.g. if we declared an array as int Data[10] we could obtain the memory address used to store the 1st item as &Data[0]).

Often of all the addresses of an array, it is the **base address** (corresponding to item [0]) that is the most useful to us. C helps us in this with the fact that, while can retrieve this using the zero index of the array (i.e. &Data[0]), if we omit the square brackets altogether we also get the base (start) address for the array.

15.1 Pointers and arrays

Now we appreciate that an array can also be considered as a series of memory addresses (the base (start) address one being a particularly useful address) we can see how we can allocate a pointer to an **individual** item in an array (note: a pointer can only hold one memory address).

As such, we can declare an array and a pointer (they must be of matching variable types) and place the start address of the array in the pointer. The code below shows the two ways of doing this (using the address of array index zero [0] and by simply using the array name).

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

int main(void)
{

    // Declare and populate an integer array
    int MyArray[10] = {2,4,6,8,10,12,14,18,20};

    // Declate an integer pointer
    int *pI;

    // Get the start address by asking for the address iof array item [0]
    pI = &MyArray[0];

    // Or, use the fact the array name on its own is the start address of the array
    pI = MyArray;

    return 0; // Exit
}
```

Listing 15.1: Setting a pointer to the start of an array [c15\pointer_array_example_1.c]

The pointer variable pI points to the 1st item in the array (the address of array index [0]), as such if we used *pI in code (remember, this means go to the memory address as held in pI and access the value) we would 'see' the value 2 (exactly the same as if we had requested MyArray[0]). The code below shows this

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

int main(void)
{
    // Declare an integer array and an integer pointer
    int MyArray[10] = {2,4,6,8,10,12,14,16,18,20};
    int *pI;

    // Get the start address by asking for the address iof array item [0]
    pI = &MyArray[0]; // Or use: pI = MyArray;

    // Display the 1st item in the array, first be accessing rhe array
    printf("The value at array item [0] is %d\n", MyArray[0]);

    // Since the pointer points to the address of the 1st item we can
    // access it as we would for a pointer pointing to any single variable

    printf("The value at the memory address held in pI is %d\n", *pI);

    return 0; // Exit
}
```

Listing 15.2: Using a pointer to access the 1st array item [c15\pointer_array_example_2.c]

This may currently seem like a limitation - we can only look at the first item in the array! There is, of course, a solution...

Pointers can be indexed like arrays

Pointers can be indexed just like an array (we drop the asterisk in this case), meaning the following

pI[0] is the same as MyArray[0]
pI[1] is the same as MyArray[5]
etc.

As such, if we were to use a loop, the loop variable could be used with the [] of either the array or the pointer variable e.g.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      // Declare an integer array and an integer pointer
8      int MyArray[10] = {2,4,6,8,10,12,14,16,18,20};
9      int *pI;
10     int i;
11
12     // Get the start address by asking for the address iof array item [0]
13     pI = &MyArray[0]; // or use: pI = MyArray;
14
15     // Use loop to display values
16     for ( i = 0 ; i < 10 ; i++ )
17     {
18         printf ("Value at index %d (direct access to the arrays) is: %d\n", i, MyArray[i]);
19         printf ("Value at index %d (access via the pointer) is: %d\n", i, pI[i]);
20     }
21
22     return 0; // Exit
23 }
```

Listing 15.3: Setting a pointer to the start of an array [c15\pointer_array_example_3.c]

A warning...

We have to ensure our pointer was set to the start of an array not an individual variable (only we can know this) - the compiler **cannot** not spot this mistake).

If you make this mistake the code will fetch values from memory locations that are NOT valid - which can have very *interesting* results!

15.2 An alternative way to move through arrays

The use of pointers allows us an additional method to move through arrays which is faster (though perhaps more complex).

Each pointer 'knows' the amount of storage used for its own type so, if we increment a pointer we move to the next memory location used to store a value (the next item in the array). This method is 'faster' and so we often use it when speed it critical.

If we had an array declared as int MyArray[10] and assigned the pointer pI to the start of the array, we know that we could access the 1st item as MyArray[0], *pI or pI[0].

To access the next item we could use MyArray[1], pI[1] - we can however 'move' the pointer to the next address using *pI++ and then retrieve the value (from *pI).

We often use this approach in loops to set values, the code below shows how this can be done.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      // Declare an integer array and an integer pointer
7      int MyArray[10];
8      int *pI;
9      int i;
10
11     // Get the start address by asking for the address iof array item [0]
12     pI = &MyArray[0]; // or use: pI = MyArray;
13
14     // Use loop to display values
15     for ( i = 0 ; i < 10 ; i++ )
16     {
17         // Set the value then use the increment operator to move the pointer
18         // to the next memory location. you can picture this as two steps: '
19         //
20         // *pI = 5 + 4*i;
21         // then
22         // *pI++;
23
24         *pI++ = 5 + 4*i; // set value at index[i] to 5+4*i
25     }
26
27   // Display the values placed in the array
28     for ( i = 0 ; i < 10 ; i++)
29     {
30         printf("%d ", MyArray[i]);
31     }
32     return 0; // Exit
33 }
```

Listing 15.4: Efficiently using pointers with arrays [c15\pointer_array_example_4.c]

One thing to note is using this method is that you will need to 'reset' the pointer back to the start if you with to then go through the array again (i.e. repeat line 13 in the above example.)

One last way to access an array using a pointer...

In fact, we can access items one other way, adding the index to the base address, e.g. to get the 5th item we can also use *(pi+4) (remember, arrays start at zero!).

At this point (perhaps not unreasonably!) you may be thinking...

> Why make this so complicated! Why do we do this?
>
> The reason is it allows to use efficiently manage memory and also to use arrays with functions... As we shall see in the next few chapters!

Chapter 16

Dynamic Memory Allocation

When allocating arrays we have, so far, always assumed that

- We know how large our array needs to be when we write the code
- There is sufficient free memory for our array

The problem is that, for both of these, it is often not the case.

To write memory efficient, stable code we need to be able to be able to allocate and free memory when the code is running (remember: memory, as we have previously used it, is not freed up until a function ends - so if we declared a huge array in main() the memory cannot be reused elsewhere in the program - even if our code no longer requires array).

16.1 How big is my array?

When developing an application to, say, record data from an experiment we could guess at the number of samples that will be read and declare an array large enough to hold them (hint: we are likely to be wrong!).

The problem here is twofold; one day more samples will need to be taken (in which case you need to re-write the code) or if the code is moved to a machine with less memory it will crash when trying to get the memory the code was originally designed to use.

Working out how large an array to be when the code is running is a relatively easy task - we just need to ask the right questions, we might ask for a single value or, perhaps, a number of questions that enable us to calculate it (e.g. if we ash 'How many samples per second' and 'How many minutes to sample for?' the number of samples is the two answers multiplied together.

At this point we need to remember that different variables take up different amount of space in memory, we do not need to know the size - we can just ask through code (indeed this is the correct was as, on different machines, some variables use a different amount of memory).

So this gives us two values

- The size of our array (how many items)
- The amount of memory for our array = size of array * size of item

Once we know this, we are in a position to request memory for our array - this time in a way that allows us to return it once we have finished with it!

16.2 Dynamically allocating & freeing memory

This again makes use of pointers, in previous sections you have seen how we can obtain the address of an existing variable (or the start of an array) and assign this to a pointer. We can then, via the pointer, access the memory location to set or retrieve values.

Dynamic memory allocation works in a similar way however this time, rather than taking the address of an existing variable we 'ask' for a block of memory of a given size.

If space is available in memory, we are provided with the base memory address - the memory is reserved for us until such time as we free (release) the memory. Should space not be available we can detect this and have our code act accordingly.

There are two functions in C that allow us to request memory, these are

- calloc: Request a block of memory for n items of size s
- malloc: Request a block of memory of a specified size in **bytes**

Both functions allocate memory and return, if memory can be reserved, the base (start) address of the memory allocated (NULL if memory could not be reserved).

The key difference is that **calloc** writes zero to all memory allocated locations which can have a speed implication (and may be unnecessary if you then intend to initialise the memory with your own values). As such, we may on occasions choose to use malloc - the size in bytes we would request would simply be the two parameters from calloc multiplied together.

16.3 The five steps of dynamic memory allocation

When wishing to dynamically allocare memory it is easiest to think of this in terms of five steps

- Declare a pointer of relevant type
- Use malloc or calloc to request the amount of memory we need.
- Check to see if memory was available to be allocated
- Make use of the allocated memory
- Free the memory (so allowing it to be reused)

16.3.1 Declare the pointer

This is the easy bit, just think of the type of array you wish to allocate (int, float, char etc.) and declare a pointer of this type.

16.3.2 Request the memory

Here is where you decide if you wish to use malloc or calloc. In some ways **calloc** is perhaps easier when learning as it 'hints' at what is required - we also get the advantage of the memory having been initialised to zero.

Regardless of your choice, the approach is the same - we pass to malloc/calloc the correct parameters and store the returned value in our previously declared pointer. If we consider dynamically allocating an array of 1000 integers we would have

```
1  {
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7
8      // Declare an integer array and an integer pointer
9      int *pData;
10
11     // Using malloc
12     pData = malloc ( 10000 * sizeof (int));
13
14     // Using calloc
15     pData = calloc ( 10000 , sizeof (int));
16
17     return 0; // Exit
18 }
```

Listing 16.1: Examples of malloc and calloc [c16\alloc_example_1.c]

Note: To switch between malloc and calloc we can simply swap the comma for an asterik!

Something to note here... the return type from malloc/calloc is **void ***, which can be implicitly converted to any pointer type - as such when using these function we are not required to typecast the value returned.

There are long (and heated) discussions on typecasting **void *** pointers, one side saying it is unnecessary (& wasteful) as they are designed to assignable to any pointer variable.

Others will state that by doing so, the compiler can spot an incompatible assignment (if suitable warnings are enabled).

The code below shows both approaches

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      // Declare an integer array and an integer pointer
8      int *pData;
9
10     pData = calloc ( 10000 , sizeof (float)); // No warning
11     pData = (float *)calloc ( 10000 , sizeof (float)); // Warning
12
13     return 0; // Exit
14 }
```

Listing 16.2: Typecasting with malloc and calloc [c16\alloc_example_2.c]

In the above example, if the developer has decided they now wish the array to be one of floats, updated calloc but forgot to change the pointer declaration.

Line 10 will not cause a warning, the memory address will stored in the pointer variable pData - problems will come later as the amount of memory allocated will not be enough (a flot required more storage than an integer).

Line 11 **will** generate a *'warning: assignment from incompatible pointer type [-Wincompatible-pointer-types'* which the programmer should investigate, they spot they did nto change the pointer type and fix the warning - which in turn will lead to stable code.

It is left to you to decide which approach you wish to follow!

16.3.3 Checking memory was available to be allocated

If malloc/calloc are able to allocate memory they return the base (start) address of the memory set aside for us to use, if not they return NULL.

As good programming practice, we should **always** check the value returned (which we will have stored in our pointer variable). If it is NULL then the code cannot (most probably) continue, some suitable error message needs to be displayed and the code terminate gracefully (or follow some other flow applicable in such cases).

This test is just a simple **if** condition added after the malloc/calloc call.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      // Declare an integer array and an integer pointer
8      int *pData;
9
10     // Using calloc (same approach malloc)
11     pData = calloc ( 10000 , sizeof (int));
12
13     if ( pData == NULL)
14     {
15         printf ("\nMemory could not be allocated - terminating");
16         return -1; // Use minus one as we did not exit successfully
17     }
18
19     // We have our memory, make use of it here!
20
21     return 0; // Exit successfully
22 }
```

Listing 16.3: Checking memory could be allocated [c16\alloc_example_3.c]

16.3.4 Using our allocated memory

At this point, you can consider the memory to have been allocated as if it were an automatically allocated array (e.g. int pData[1000])

You can access individual items as (say) pData[0], pData[25], or using the other pointer based approaches covered in chapter 15.

16.3.5 Freeing up memory

Once we have finished with the memory it is **very important** that we free it (making it available for storing other variables, the next time we use malloc/calloc etc.).

To do this we use the function *free*, passing to it the pointer variable which holds the base address of memory previously allocated.

This then leads us to the completed version of our example program which requests for memory (line 11), checks it was allocated (line 13) and then frees up the memory once it is no longer needed (line 22).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6
7      // Declare an integer array and an integer pointer
8      int *pData;
9
10     // Using calloc (same approach malloc)
11     pData = calloc ( 10000 , sizeof (int));
12
13     if ( pData == NULL)
14     {
15         printf ("\nMemory could not be allocated - terminating");
16         return -1; // Use minus one as we did not exit sucesfully
17     }
18
19      // We have our memory, make use of it here!
20
21     // Free up the allocated memoey
22     free (pData);
23
24     return 0; // Exit sucesfully
25 }
```

Listing 16.4: Allocating, checking and freeing up memory

free: an important note

Memory allocated using malloc/calloc is **not** released when a function exits, only the memory assigned to the pointer variable declared to hold the address is released.

This can cause **huge** problems of 'memory leaking', where over time blocks of memory are allocated but never freed.

It tends most often to occur when memory is allocated in a function and set to be released at the end however the function exits early and the **free** statement(s) are never executed.

The example below demonstrates how a memory leak may occur

```
1  void DoSomeWorkInC ( int n, int t)
2  {
3          float *Data
4
5          Data = calloc( n, sizeof (float))
6          if ( Data == NULL)
7          {
8              printf("No memory for function, exiting\n");
9              return -1 // -1 will mean something to the caller
10         }
11
12         // A simple test which may equate non-zero (true)
13         if ( t > n )
14         {
15             printf ("Test condition failed, exiting function");
16             return -2; // -2 will mean something to the caller
17         }
18
19         free (Data); // Free up the allocated memory
20 }
```

Listing 16.5: How memory leaks

In the above example, the memory is correctly allocated (with checking) however, if the test condition fails (if the value of **t** passed to the function is greater than **n** the function will exit (return to the point where it was called) **without** the **free (Data)** statement being executed.

This means that the memory allocated (line x) has not and cannot be released as the pointer variable is automatically destroyed when the function exits (like any automatic variable).

The problem would have been avoided if the free statement was also included with the code executed if the **if** statement equated non-zero (true).

```
1    // A simple test which may equate non-zero (true)
2    if ( t > n )
3    {
4      printf ("Test condition failed, exiting function");
5
6      free (Data); // Free up the allocated memory within if statement
7
8      return -2; // -2 will mean something to the caller
9    }
```

Listing 16.6: Fixing the leak

Now we know how to dynamically allocate memory, we can combine this with our knowledge of functions to develop memory efficient, robust code!

NOTE: It is possible to allocate multidimensional arrays using the approaches outlined in this chapter however it is outside the scope of this beginners course.

Details (and examples) on on how to do this can be found on-line.

Chapter 17

Functions (Part 3)

As well as passing individual variables to functions there are many times we may wish to pass an array of values.

When declaring a function to which we wish to pass an array we can do this one of two ways, using a pointer to receive the start address of the array, e.g.

```
int DoSomething ( float *Data );
```

While this is perfectly correct, it does not immediately draw to our attention that 'Data' will be an array, as such we often write the function prototype as

```
int DoSomething ( float Data[] );
```

This works as, in C, declaring an array without a size is the equivalent of defining a pointer.

17.1 Accessing array data in functions

As we are passing the array data via a pointer (in fact there is no way not to, C will always take this approach!) this does mean that within the function we can change the contents of the **original** array.

In many ways this is a very useful thing to be able to do - we can use functions to populate arrays, display contents etc. all of which make for better and more stable code.

One other point to note is that when passing arrays to function we (generally) need to pass the size of the array - this cannot be determined from the pointer.

By way of an example of how we can both access items in an array passed to a function **and** change them, the following code example will

- Declares an array of integer if size 10 in main()
- Passes the array to a function which populates the array
- Passes it to a second function which displays the values in the array.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Simple function to populate an integer array
5  void PopulateTheArray ( int Size, int ArrayData[])
6  {
7      int i; // Variable to use in our loop
8
9      for ( i = 0 ; i < Size ; i++)
10     {
11         ArrayData[i] = 2*i + 1; // Treat it like a normal array
12     }
13 }
14
15
16 // Simple function do display contents an integer array
17 void DisplayTheArray ( int Size, int ArrayData[])
18 {
19     int i; // Variable to use in our loop
20
21     for ( i = 0 ; i < Size ; i++)
22     {
23         printf ("Item_%d_of_the_array_contains_%d\n", i, ArrayData[i]);
24     }
25 }
26
27 // Main () - execution starts here
28 int main (void)
29 {
30     int Data[10];
31
32     // Pass the size of the array and the array to our function -
33     // remembering that the array name on its own is the base address,
34     // and so is the same as passing &Data[0]
35
36     PopulateTheArray(10, Data);
37     DisplayTheArray(10, Data);
38
39     return (0); // Exit indicating sucess
40 }
```

Listing 17.1: Passing arrays to functions [c17\arrays_to_functions_example_1.c]

To make use of all we have learnt about allocating memory for arrays, we now change this program such that it asks the user how large the array should be.

It then allocates the required amount of memory (with checking) before using the same functions to populate & then display the contents of the array.

The final step is then to free up the allocated memory.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Simple function to populate an integer array
5  void PopulateTheArray ( int Size, int ArrayData[])
6  {
7      int i; // Variable to use in our loop
8
9      for ( i = 0 ; i < Size ; i++)
10     {
11         ArrayData[i] = 2*i + 1; // Treat it like a normal array
12     }
13 }
14 // Simple function do display contents an integer array
15 void DisplayTheArray ( int Size, int ArrayData[])
16 {
17     int i; // Variable to use in our loop
18
19     for ( i = 0 ; i < Size ; i++)
20     {
21         printf ("Item %d of the array contains %d\n", i, ArrayData[i]);
22     }
23 }
24 // Main () - execution starts here
25 int main (void)
26 {
27
28     int iSizeForArray;
29     int *pData; // A pointer to hold the base address of out array
30
31     // Ask for the size of the array and store result
32
33     printf("\nPlease enter the size of the array to dynamically allocate");
34     scanf ("%d", &iSizeForArray);
35
36     // Use calloc with checking
37     pData = calloc ( iSizeForArray, sizeof (int));
38
39     // Check we got the memory
40     if ( pData == NULL)
41     {
42         printf ("\nSorry, I could not allocate the memory, bye!");
43         return -1;
44     }
45
46     // Pass the size, iSizeForArray) and the pointer created
47     // which points to the start of the sucesfully allocated memory
48
49     PopulateTheArray(iSizeForArray, pData);
50     DisplayTheArray(iSizeForArray, pData);
51
52     free (pData); // Free up the memory before exiting
53
54     return (0); // Exit indicating sucess
55 }
```

Listing 17.2: Bringing it all together [c17\arrays_to_functions_example_2.c]

Chapter 18

Using Files

Often, when developing applications, we may need to either save the output to a file or read data from a file to enable it to be processed.

In C there are two types of files we work with, namely

- Text files
- Binary files

Each have their own advantages and disadvantages (based on the requirements of our application), some of which are detailed in table 18.1. Knowing these, it is up to us as analyst/programmers to determine the most appropriate to use.

File Type	Advantages	Disadvantages
Text	Can be viewed in a editor, listed, printed Can be read by different machines	Tend to be bulky Must be read in sequence
Binary	Much smaller for the same amount of data Can be randomly accessed	Byte ordering can be a problem

Table 18.1: Comparing Text and Binary files

Note: Text files can be written so they can be read non-sequentially but it is a very *messy* process!

18.1 The common tasks when using files

When working with files, many of the steps are the same. We first need to open file (with suitable checking) and must ensure we close them when they are no longer required to be accessed.

What differs between the two file types is the methods for reading and writing information.

18.1.1 Declaring a file pointer

When we wish to open a file in C (using the function **fopen**), we first need to declare a **stream pointer** that is able to connect out application to a file. This done through the use of a specific variable type **FILE** which is defined in **stdio.h** (it is one of the few times you will use capital letters when declaring a variable in C).

This special type contains all the information required to access a file - we do not need to access this (in fact it is recommended not to).

To declare a stream variable variable that we can then **point** to a file we use the syntax

FILE *fPtr;

Note that we can have multiple files open at any time, to define (say) one file for input and another for output we can simply write

FILE *fIn, *fOut;

18.1.2 Opening a file

In C, when we wish to open a file there are two parameters we need to pass to the function that performs this task for us, they are

- The name of file to open (including path if appropriate)
- The mode in which we wish to open the file

The file name

This is simply the name of the file we wish to open, it can be fixed (in which case we provide it within double quotation marks) or, if we area asking the use to provide it at runtime, held in a string.

Note that, if no path is provided the file will be considered to be in the same location as the application.

A note on 'fixed' path and file names

If providing a path as part of a fixed file name in code, we need to be aware of how to write these (this is not a problem if these are provided as a string variable).

In C the \symbol is used to indicate a control sequence (for example \n means newline). This can cause problems with writing file names as a path of

"c:\new data\test sample.dat"

Would have the \n as a new line and \t as a tab symbol!

To get round this, there are two options

- use \\in place of a single \
  e.g. "c:\\new data\\test sample.dat"
- replace the \with a /
  e.g. "c:/new data/test sample.dat"

File open modes

In addition to specifying the file to open, we also need to state **how** we will be opening the file (e.g. for reading, writing). This is done by an additional parameter passed to the open file function.

This parameter is defined as a string as, on occasions, multiple characters are used to define the mode.

Table 18.2 lists the the most common used to open a file (a search on-line will provide a much longer list, specifically on how to open a file in update mode which itself requires additional consideration as to how we access the file).

Mode	Descriptions
"r"	Open a text file for reading
"w"	Open a text file for writing if the file already exists, the original contents is deleted
"a"	Append; move to the end of an existing text file if no file exists, create a new file (as if "w" were the mode)
"rb"	Open a binary file for reading
"wb"	Open a binary file for writing if the file already exists, the original contents is deleted
"ab"	Append; move to the end of an existing binary file if no file exists, create a new file (as if "wb" were the mode)

Table 18.2: Modes in which a file can be opened

18.1.3 Closing files

Once we have finished accessing a file we **must** close the file - this writes any remaining information to file and makes it available to others processes.

We must remember to that, if we exit early from a function (including main) that to close any files previously opened (in the same way we free allocated memory to avoid memory leaks).

The C function for closing a file is **fclose**, it takes a single parameter - a stream pointer. It returns zero if successful, **EOF** (declared in **stdio.h** if errors are detected,

18.1.4 Examples of opening a file, with error checking (and then closing it)

As with memory allocation, good programming involves checking that actions can be completed - the same is true when opening files. We do this by looking, as with malloc/calloc the value returned and comparing this to NULL.

If NULL is returned there was an error opening the file and, as such, we cannot continue; any non-NULL value indicates the file was sucesfully opened.

The C function we use to open a file is **fopen** - it is formally defined as

```
FILE *fopen(const char *filename, const char *mode);
```

The value it returns we place in our declared variable the file name and mode are the parameters previously detailed.

Let us now consider examples of opening (with error checking) and closing files.

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

// Main () - execution starts here
int main (void)
{

    // Declate file stream variables
    FILE *fInput, *fOutput, *fRecords;


    // Try and open the text "sample.txt" (in the current directory) file for reading
    fInput = fopen ("sample.txt", "r");

    // Check we were able to open the file
    if ( fInput == NULL)
    {
        printf ("\nthe_file_could_not_be_opened");
        return -1; // Exit as unsuccessful
    }

  fclose (fInput); // Close the file

    // Try and open the binary "samples.dat" (in the current directory) file for writing
    // if a file of this name already exists it will be deleted
    fOutput = fopen ("samples.dat", "wb");

    // Check we were able to open the file
    if ( fOutput == NULL)
    {
        printf ("\nthe_file_could_not_be_opened");
    return -1; // Exit as unsuccessful
    }

  fclose (fOutput); // Close the file

    // Open, for appending, the text file "records.txt". If the file does not already
    // exists, a new one of this name will be created (as if "w") were the mocde
    fRecords = fopen ("records.txt", "a");

    // Check we were able to open the file
    if ( fRecords == NULL)
    {
        printf ("\nthe_file_could_not_be_opened");
    return -1; // Exit as unsuccessful
    }

  fclose (fRecords);


    return (0); // Exit indicating sucess
}
```

Listing 18.1: Open file examples [c18\file_open_example.c]

Note: In practice, we would be accessing the files before we close them!

Now we now how to open and close files, let us look at how we access them.

18.2 Text files

Text files are perhaps the easiest to work with, both in the fact we can 'view' the contents but also as the commands to read and write to files are almost identical to those we use to write to the screen (e.g. printf) or read from the keyboard (scanf).

Commands to access files almost invariably start with the letter **f** - giving us

- fprintf : Write information to a file
- fscanf : Read information from a file
- fputs : Write a string to a file
- fgets : Read a string from a file

The key difference when using file functions is that we must provide the stream pointer as a parameter (generally, but not always, as the 1st parameter).

If we assume a file pointer fPtr has been successfully declared and used to open a file (in the correct mode appropriate to reading/writing), the table below shows how the function to access files relate to the 'non-file' equivalents. Assume too all other variables are also declared.

Screen/Keyboard	File
printf ("Hello World\n")	fprintf (fPtr, "Hello world\n")
printf ("The value of x is %d\n",x)	fprintf (fPtr, "The value of x is %d\h n",x);
scanf ("%d", &d)	fscanf (fPtr, "%d", &d)
puts (MyString);	fputs (fPtr, MyString)
gets (MyString)	fgets (MyString , 100, fPtr);

Table 18.3: Functions to read/write text files

Note: fgets is different in that the number of characters to read must be specified (to avoid a buffer overflow), the file pointer also comes at the end

There are many other functions for reading/writing to text files - a quick on-line search will provide help on these.

18.2.1 Text file example

The following example shows how a text files can be used. A new file is created to which the values 1-10 inclusive are written, if the then closed, re-opened in 'read' mode and the numbers read in and displayed on the screen.

note in this case we know how many items to read from the files, this is not always the case. Solutions to this are presented in section 18.5.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Main () - execution starts here
5  int main (void)
6  {
7      // Declate file stream variables
8      FILE *fInput, *fOutput;
9
10     // Other variables needed
11     int i,d;
12
13     // Try and open the text "sample.txt" (in the current directory) file for writing
14     fOutput = fopen ("numbers.txt", "w");
15
16     // Check we were able to open the file
17     if ( fOutput == NULL)
18     {
19         printf ("\nthe file could not be opened for writing, exiting");
20         return -1;
21     }
22
23     // Use a loop to write values to the newly created file
24     for ( i = 1 ; i <= 10 ; i++)
25     {
26         fprintf (fOutput, "%d\n", i);
27     }
28
29     // And close the file
30     fclose (fOutput);
31
32     // Try and open the binary "numbers " (in the current directory) file for reading
33
34     fInput = fopen ("numbers.txt", "r");
35
36     // Check we were able to open the file
37     if ( fInput == NULL)
38     {
39         printf ("\nthe file could not be opened for reading, exiting");
40         return -1;
41     }
42
43     // And close the file
44     fclose (fInput);
45
46     // Read, line by line the 10 values written into variable d
47     // and then display the contents of d on the screen
48     for ( i = 1 ; i <= 10 ; i++)
49     {
50         fscanf (fInput, "%d", &d);
51         printf ("Value read from file %d\n",d);
52     }
53
54     return (0); // Exit indicating success
55 }
```

Listing 18.2: Writing and reading text files [c18\text_file_example.c]

18.3 Binary files

Binary file access works by copying blocks of memory to/from files without first formatting it (as we do with text input/output).

As such, it is much faster than working with text files however it does bring with it some points we must note.

- We cannot directly 'view' the information with (say) an editor
- The ordering of the bytes used to store the information can differ between systems

The first of these we can overcome by having software that can load binary data and then display it in a human readable format (e.g. importing the binary data into Matlab or perhaps writing our own application to convect between binary and text files).

The byte ordering problem can be more challenging to deal with. It arises as, when writing numbers to memory the order in which the bytes used for storage are arrange can be in one of two orders (starting at the base address).

- From low byte to high byte
- From high byte to low byte

This becomes a problem where we move a binary file from a system using one style to another using the other - the numbers become 'jumbled'.

As long as we are aware of this problem there are techniques for overcoming it (some of which will be detailed in section 19.5), some software packages (e.g. Matlab allow you to specify the byte ordering as parameters when opening file to overcome this problem).

18.3.1 Reading and Writing to/from binary files

The most frequently used commands to perform this are almost identical, the only difference is the actual name of the functions - which are

- fwrite: write data to a binary file
- fread: read data from a binary file

The formal definitions for the functions (from **stdio.h**) are

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream)

Let us look at each in turn

void ptr

This parameter is the address of an existing variable (or base address of an array if we reading/writing multiple items).

void size_t size

This is the size, in bytes, of the type of item we are are reading - we again use the sizeof function determine this value (to allow for the fact that, on different systems, different numbers of bytes may be used for storage).

void size_t nmemb

This is the number of items we wish to read/write. For a single variable it will be 1 (one), for an array it would be the size of the array.

FILE *stream

This is the file pointer we have previous declared and which has been assigned a value by **fopen**.

The return value

Note that both fread/write return a value - this is the number of bytes successfully read/written. We can examine this to determine if the action was carried out successfully, taking any required action if not.

18.3.2 Binary file examples

The example below provides an example of reading and writing to binary files.

Once the file has been successfully opened (in the required mode), first a single integer and then a complete int array of size 10 (pre-populated with the values 1-10) is written to the file. The file is then closed and the values read back and displayed on the screen.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Main () - execution starts here
5  int main (void)
6  {
7
8      // Declate file stream variables
9      FILE *fInput, *fOutput;
10
11     // Other variables needed
12     int i;
13     int SampleArray[10] = {1,2,3,4,5,6,7,8,9,10};
14     float f = 23.4;
15
16     // Try and open the binary "numbers.dat" (in the current directory) file for writing
17     fOutput = fopen ("numbers.dat", "wb");
18
19     // Check we were able to open the file
20     if ( fOutput == NULL)
21     {
22         printf ("\nThe file could not be opened for writing, exiting");
23         return -1;
24     }
25
26     // Write out a single float to the binary file
27     fwrite ( &f, sizeof(float), 1 , fOutput);
28
29     // Now the entire array on one go
30     fwrite ( SampleArray, sizeof(int), 10 , fOutput);
31
32     // And close the file
33     fclose (fOutput);
34
35
36     // Try and open the binary "numbers.dat" (in the current directory) file for reading
37     fInput = fopen ("numbers.dat", "rb");
38
39     // Check we were able to open the file
40     if ( fInput== NULL)
41     {
42         printf ("\nthe file could not be opened for reading, exiting");
43         return -1;
```

```
44      }
45
46      // Write in a single float from the binary file into f
47      fread ( &f, sizeof(float), 1 , fOutput);
48
49      // Now read the entire array on one go
50      fread ( SampleArray, sizeof(int), 10 , fOutput);
51
52      // Display the values read from the file on the screen
53      printf ("The value read into f is %f\n", f);;
54      for ( i = 0 ; i < 10 ; i++)
55      {
56          printf ("Item %d of the array contains %d\n",i, SampleArray[i]);
57      }
58
59      // And close the file
60      fclose (fInput);
61
62
63      return (0); // Exit indicating sucess
64  }
```

Listing 18.3: Writing and reading binary files

18.4 Reading in a specific item from a file

On occasions we may wish to read a specific item from a file - this could be the 50th item in the file or the last one.

With text files we have (almost) no alternative other then to open the file, read each line until we find the item we required. This is wasteful as then to obtain another specific item, we need to return to the start of the file and repeat the process.

When working with binary files they is a solution to this problem. In a binary file each item of a given type takes up the same amount of space in the file, regardless of its contents.

As such, if we know the 'block' size we can calculate how far along in the file the item we wish to read will be located.

The C function that aids us in this is **fseek**, formally defined as

int fseek(FILE *stream, long int offset, int whence);

It allows us to move to a specific point in a file (whose file stream is passed as the 1st parameter) based on two criteria

- offset : the number of bytes to move by (can be positive or negative)
- whence : the point in the file from where the movements is to be based

There are three possible values for whence which are defined in **stdio.h**, they are

- SEEK_SET : Move from the start of the file
- SEEK_CUR : Move from the current position
- SEEK_END : Move from the end of file

When using SEEK_SET clearly the value of offset must be positive and no more than the size of the file in bytes, likewise then using SEEK_END the value of offset must be positive and no

more than the size of the file. Values of zero for the offset in both cases are valid (allowing us to move to the start and of the file respectively).

An alternative to using an offset of zero with SEEK_SET to move to the start of a file is to use the **rewind** function - all that needs to be passed to this is the file pointer, e.g. (assuming fPtr to have previously been declared)

rewind (fPtr);

is eqivalent to

fseek (fPtr, 0L, SEEK_SET);

18.5 How big is my file - method 1

If we do not know how many items there are in a file, we may just have to read item by item until we get to the end - counting as we do so.

C provides a function that returns a value indicating if the end of file has been reached, it returns

- 0 if the end of file has **not** been reached
- non-zero when the end of file is encountered

We most often use this in a **while** loop, reading values - either using them directly or counting/analysing them (perhaps to then rewind the file and to re-read them into a dynamically allocated array).

As the value returned by feof is zero **until** the end of file is reached we need either to compare to zero or, use the not operator ! to invert the value. The snippet of code below show how we can use feof to read to the end of a previously opened file

```
1  while ( !foef (fInput)) // Invert the result 0->1, 1->0
2  {
3    fscanf(fInput, "%d", &value);
4  }
5
6  while ( foef (fInput) == 0) // Loop while value returned is zero
7  {
8    fscanf(fInput, "%d", &value);
9  }
```

Listing 18.4: Read to the end of a file

18.6 How big is my file - method 2

Once we are at the end of a file (either by reading to the end or using fseek) we can ask for the current file position in bytes - effectively the size of the file.

The function for this ftell, formally defined as

long int ftell(FILE *stream);

If we are working with a binary file (or very fixed format text file), we can calculate the number of items in a file by dividing the file size (from **ftell** by the item size, e.g. if we knew we had opened a binary file to which floats had been written, we could determine the size as (assume fPtr to have been declared and the file opened correctly etc.)

NoElements = ftell(fPtr) / sizeof (float);

We might even write a small function which, when passed a file pointer & the size of each item moves to the end, calculates the number of items, rewinds the file and then returns the calculated value. Such code might resemble

```
1 long int ObtainItemsInFile ( FILE *fPtr, int ItemSize)
2 {
3     long int Items;
4
5     fseek (fPtr, 0L, SEEK_END); // Move to the end of file
6     Items = ftell(fPtr) / ItemSize; // Calculate number of items
7     rewind (fPtr); // Move back to the start
8     return (Items);
9 }
```

Listing 18.5: Function to calculate items in a binary file

This could then be called in main

ItemCount = ObtainItemsInFile (fData, sizeof(int));

18.7 How big is my file - method 3

Often the easiest way to find this it to put the information at the beginning of the file! Such information is referred to as a **header**.

File headers can be as simple or as complex as required (from a single value then indicates the number of items in a file to information on an image including the colour palette).

An application would read this information, using the information to determine how the code should then operate (e.g. allocating memory of the correct size, perhaps initialising a piece of equipment).

Where headers get very complicated we often define a collection of variables to hold these - such a collection is referred to as a structure, these being covered in section [X].

With text files are are (generally) committed to writing the header information before writing the remainder of the information - this does of course require us to have all this information available (which is not always the case).

When working with binary files, it is possible to write a *dummy* header that is later overwritten (by rewinding the file back to the start and writing the values again - this works as the number of bytes is the same so it does not corrupt the remainder of the file).

Do remember (especially when working with binary files) that if you then wish to seek for a specific item to take into account the size of the header! If we wished to select the 10th float from a binary file with no header we would move forward 10*sizeof(float) bytes in the file, i.e.

offset = distance to required item
offset = 10*sizeof(float)

If we had a header of 5 floats (perhaps items, max & min for x and y data) the offset to move would be calculated as

offset = header size + distance to required item
offset = 5*sizeof(float) + 10*sizeof(float)

Then use

fseek (offset, SEEK_SET, fPTR);

Chapter 19

Advanced Data Types in C

This chapter looks at some of the more advanced data types we can use when programming in C, they help enable us to write efficient code that is (ideally) easier to manage.

19.1 defines

When developing code, we aim to make our code both as readable and maintainable as possible. Much of this we can achieve through sensible variable and function names, clear commenting and good layout.

Another technique is to develop 'labels' which we can use in our code, such examples might be mathematical constants (e.g. M_PI) or NULL (defined in stdlib.h).

We can define our own through the use of the #define compiler directive - this allows us to provide an 'alternative' that we use in code that, prior to compilation, is substituted in our code.

the format is

#define **label** *thing it replaces*

e.g. using M_PI as an examples (taken from math.h)

#define M_PI 3.14159265358979323846

every time M_PI is seen in code, it is replaced with 3.14159265358979323846

WARNING!

Do NOT add a semicolon on the end of a #define as this will also be substituted!

This can be a huge problem to track down as the code invariably looks OK until you then check out the #define!

Another time we can use #define statements is to define things we may need to change across code (e.g. a configuration parameter).

Consider the case of system we have developed code for where a button up/down we read 0/1 respectively. The hardware is now redesigned and the opposite is true (up=1, down=0).

If we have used #define for each, we need only change these and recompile rather than having to check all occurrences of 0 or 1 in code (which could take a very long time!).

Consider the example code below, we first define UP and DOWN as 1 and 2 respectively, we then use these labels in code

```
1  #define UP 1
2  #define DOWN 2
3
4  int main()
5  {
6      int i = 1;
7
8      if (i == UP )
9      {
10         // Do something
11     }
12
13     if ( i == DOWN)
14     {
15         // Doe something else
16     }
17     return 0;
18 }
```

Listing 19.1: Example of using #define [c19\define_example.c]

What is actually compiled is

```
1  #define UP 1
2  #define DOWN 2
3
4  int main()
5  {
6      int i = 1;
7
8      if (i == 1 )
9      {
10         // Do something
11     }
12
13     if ( i == 2)
14     {
15         // Doe something else
16     }
17     return 0;
18 }
```

Listing 19.2: The code as it is compiled

A second warning!

As this is a simple 'find and replace' approach the compiler cannot check if defines are re-used e.g.

```
#define UP 1
#define DOWN 2
```

If you used these in an *if* statement there would be no error but your code **would not** execute as expected (a swtich-case statement woudld throw a duplicate case warning - **if** you have all warning set & you still need to check them!).

19.2 Enumarations

An enumerated variable type is a method of creating a series of integers which have sequential values. The have advantages over other methods in that, if we add an additional item to the list the numbers allocated automatically adjust so, as long as we use the labels we can never repeat a value.

They are particularly useful in switch-case constructs where we can then use these labels. e.g if we defined an enumerated type DOW which we then defined as

enum DOW sun, mon, tue, wed, thu, fri, sat, sun ;

We can then use these labels within a switch-case construct as the example below shows

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum DOW { sun, mon, tue, wed, thu, fri, sat } ;
5
6  // Main () - execution starts here
7  int main (void)
8  {
9      enum DOW day;
10
11     /* Code that get a value for 'day' */
12     day = tue;
13
14     switch (day)
15     {
16         case sun : printf ("Sunday\n") ; break ;
17         case mon : printf ("Monday\n") ; break ;
18         case tue : printf ("Tuesday\n") ; break ;
19         /* etc. */
20     }
21     return (0); // Exit indicating success
22 }
```

Listing 19.3: Defining a enumerated type [c19\enum_example.c]

Note: A 'good' compiler is able to warn where enum types have been missed, i.e. for the example above the compiler noted no cases were provided for wed, thu, fri or sat!

19.3 static variables

Normally, when we declare variables within a function these are **automatically** managed - memory is allocated at the point the variable is declared and release when the function exits.

We can however make a variable remain in memory after a function exits - and so available to the function if it is again called, we this by defining the variable as **static**. To do this we simply prefix the variable type with the keyword **static**,

static int k = 0;

Note; Static variables, unlike others are given are initialised to zero (NULL for pointers), we can however provide an alternative value is this is appropriate to our code.

These provide a better alternative to global variables as their scope is **ONLY** the function in which they are declared.

In the following example the value of k is declared and initialise ONCE (line 6) in the function, following calls to the function make use of this value - in this example it therefore provides a count of the number of times the function has been called.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void DisplayHelloWorld (void)
5  {
6      static int k = 0; // Counter for how many times the function is called
7
8      printf ("Hello World\n");
9
10     // Increment counter and display value
11     k = k + 1;
12     printf ("I have now said this %d times\n",k);
13 }
14 // Main () - execution starts here
15 int main (void)
16 {
17     int i;
18
19     // Loop calling out function 10 times
20     for ( i =0 ; i < 10 ; i++ )
21     {
22         DisplayHelloWorld();
23     }
24
25     return (0); // Exit indicating success
26 }
```

Listing 19.4: static variable example [c19\static_variable_example.c]

19.4 structs

While we can write much of our code using the many variable types provided in C, occasionally we may wish to generate data structures of that group data into meaningful units.

We do this via the C process of creating a structure, by which we combine variables together into a single new variable type (the items within being referred to as members). The size of a struct is the sum of the sizes of each member.

You can picture this as a business card for a company - the definition of the card is the same (it will contain the same items [members] such as name, role, phone number, email etc.). It is possible to hand across an card or to retrieve a single items from it.

To define a structure we use the C keyword **struct**, an example of a structure to hold x, y and z data float items is below

```
1 struct Coordinates
2 {
3     float x,y, z;
4 };
```

Listing 19.5: Defining a structure

Note that this has not declared any variables, simply defined a template against which variables can be declared.

These definitions should not be placed within functions, rather at the top of code (or even in their own header file(s)).

In our code we then declare variable of this struct type as

struct Coordinates Points1, Points2;

Note: it is permitted to omit the **struct** in declarations however its presence does help with readability of code.

Once we have declared a variable to be of a struct type we can manipulate them either as complete objects, e.g. copying one to the another (which copies each member e.g.

Points1 = Points2

To access a member item, use the the dot operator, for example to obtain the value of the member x from Points1 and store it in a variable p we would use

p = Points1.x

To set the value of y member of Points1 to 3.1 we would use

Points1.y = 3.1

We can declare arrays of structures as we would for any variable type in C, the syntax is the same e.g.

struct Coordinates PointsArr[10];

To access a specific member of a specific item in the array, we use the form

p = PointsArr[3].x
PointsArr[3].y = 3.1

When are struts useful

There are a few times when you may find structs particularity useful, two good cases however are

- Passing large numbers of parameters to functions
- File headers

structs with functions

If you have a function that requires a large number of parameters we could define this as a 'typical' function as a list of type/argument parameters however this could start to get very time consuming - especially if the parameters are then passed to other functions.

A neater approach is to declare a structure that contains all the various parameters and pass this to the function (this has the added advantage that, to add another parameter, we only need to add this to the struct and all functions intermediately have access to this additional parameter.

We can also use this method to make the use of memory more efficient, if instead of passing the struct itself (so copying all the parameters - of which there may be many), we pass a pointer to the structure a single variable provides access to all the items (this is also faster which is an added bonus).

structs with files

If we need to write a file header that is complex in nature (many parameters of different types), we could do this one item at a time (using *frwite*) however the process would be somewhat tedious to code - as would the code to read the individual items back from a file.

If we are required to add/remove items from the header format we need to re-code both the writing/reading which increases the workload (and, if we omit the changes for one half of the process will lead to very odd results).

To overcome this we often define a struct that contains all the header information - this can then be written or read as a single item, greatly simplifying the process.

19.5 Unions

A definition of the members for a is identical to that to a struct, simply replace **struct** with **union**.

The difference is that in a **struct**each item is in its own area of memory (the base address of a struct is the address of the 1st item).

In a union, all member items have the same address - as such when writing one value others are also overwritten. The size of a struct is therefore that of the largest member in the union.

We use unions to allow manipulation of bytes (e.g. to swap the order of bytes) or where memory is very restricted, we can use a union to store different types of variables in the same memory space - just one at a time.

Chapter 20

Compiler Preprocessor Directives

Prior to code be compiled it is automatically run through a preprocessor which looks at the lines starting with a # symbol.

20.1 #include

One preprocessor directive which which have already seen extensively in code, this takes the contents of another file and 'inserts' it at the point where the #include statement is written.

20.2 Macros (#define)

A macro (here) means a segment of code which is replaced by the value of the macro (essentially a find & replace)

There are two variants we can make use of when developing code, these are

- Object-like
- Function-like

20.2.1 Object-like

The simplest of these if the #define statement where text is substituted replacing the identifier with the value, we have seen this for the definition of M_PI (see section 19.1)

20.2.2 Function-like

It is also possible to define simple functions which are then (again) substituted in the resulting code e.g.

```
#define MIN(a,b) ((a)¡(b)?(a):(b))
```

Which we could then use in code as shown below

```
1 #include <stdio.h>
2 #define MIN(a,b) ((a)<(b)?(a):(b))
3
4 int main(void)
5 {
6     printf("The minimum value of 10 and 20 is: %d\n", MIN(10,20));
7     return 0;
8 }
```

Listing 20.1: Macro object-like example [c20\macro_function_example.c]

When run, the output will be

```
The minimum value of 10 and 20 is: 10
```

20.3 Formatting directives

It is possible to use use preprocessor dirctoves as we would if/else statements - this allows us to include or omit code from compilation.

This can be particularly useful if we wish to have a 'debug' version where additional information is displayed when the code is executed or a 'demo' version which has reduced features yet is based on the same code.

We do this by looking to see if a macro has been defined (or not.

Consider the following example

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  #define DEBUG_ON 1
5
6  int main(void)
7  {
8  #ifdef DEBUG_ON
9    printf("Debug mode - about to do something\n");
10 #else
11   print("Running in standard mode");
12 #endif
13
14 return 0;
15 }
```

Listing 20.2: Formatting directives [c20\formatting_directive_example.c]

As we have *defined* DEBUG_ON , the '#ifdef' condition will be true so the code on line 9 will be included in that to be compiled, the code on line 11 will be omitted as that condition if false.

As such, we can consider the code to be

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  #define DEBUG_ON 1
5
6  int main(void)
7  {
8
9    printf("Debug mode - about to do something\n");
10
11   return 0;
12 }
```

Listing 20.3: Formatting directives - what will be compiled

If we remove the #define line then the #else condition will be met and the code would compile as if written below

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  #define DEBUG_ON 1
5
6  int main(void)
7  {
8
9    print("Running in standard mode");
10
11   return 0;
12 }
```

Listing 20.4: Formatting directives - what will be compiled (if not defined)

We can do the opposite of the above using the #ifndef preprocessor directive which checks to see it the macro is **not** defined

In addition to #ifdef we also have a #if directive which can be used much as the above but makes a conditional test, e.g.

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  #define DEBUG_ON 1
5
6  int main(void)
7  {
8  #if DEBUG_ON == 1
9    printf("Debug mode - about to do something\n");
10 #else
11   print("Running in standard mode");
12 #endif
13
14   return 0;
15 }
```

Listing 20.5: Conditional directives [c20\conditional_directove_example.c]

In the above, as the condition is met, line 9 would be compiled, line 11 omitted, giving

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  #define DEBUG_ON 1
5
6  int main(void)
7  {
8
9    printf("Debug mode - about to do something\n");
10
11   return 0;
12 }
```

Listing 20.6: Conditional directives - Lines to be compiled

Chapter 21

Command Line Arguments

While we often wish to interact with programs, there are times when it is convenient to pass input directly the a program at the point - e.g. if we had a a program that converts a JPEG file to a GIF to provide the files names at the point when we execute the applications, perhaps running it as

ConvertJpegToGif MyPhoto.jpg MyPhoto.gif

We achieve this in C through the use of a modified main() which is able to parse input directly into our applications (think of them as 'parameters' we pass to our main() when we start execution).

21.1 A new version of main

The first change we need to make is to use a new form of main

int main (int argc, char *argv[])

Let us look at the two parameters

21.1.1 int argc

This is a count of the command line parameters.

It must at least 1, as the program name its self is classed as a command line parameter.

21.1.2 char *argv[]

This is a pointer to an array of strings that contain the parameters

As with all arrays, it starts at zero, the zero element is the program name so to display the 1st value (the program name) we would use

printf ("%s",argv[0]);

Note: Each parameter is a string so, if numerical values are passed we need to extract them (covered in section 21.3).

21.2 Using parameters in our application

When passing parameters we need to consider which are required and if there are any that are optional - for example, if we consider out JPEG to GIF example we must provide the input and output file names, we *might* wish to include an additional scaling factor (perhaps both x & y).

This leads us to two approaches

21.2.1 The simple approach

We write code that accepts the parameters in a specific order, optional parameters can be supplied but according to specific rules e.g.

ConvertJpegToGif input oupput x_scale y_scale

Where x_scale and y_Scale are optional

We can check the number of parameters passed (from **argc**) to ensure it is at least 3 (program name, input and ouput).

If we have 4 parameters we know that the x_scale has been provided, a value of 5 means both x_scale and y_scale have been provided

The problem however is that we cannot supply y_scale unless we provide x_scale

21.2.2 The 'general' approach

Here we use 'flags' to indicate the parameter being passed - this approach is very flexible but does require considerably more code (it is something you write once and reuse!).

Using this approach we would run our example as (say)

ConvertJpegToGif -i input -o oupput -xs x_scale -ys y_scale

The advantage here is that (1) we can provide the inputs in any order, (2) optional parameters can be selected as required, so we could have

ConvertJpegToGif -i MyPhoto.jpeg -o MyPhoto.gif

ConvertJpegToGif -i MyPhoto.jpeg -o MyPhoto.gif -xs 10

ConvertJpegToGif -i MyPhoto.jpeg -o MyPhoto.gif -ys 10

ConvertJpegToGif -i MyPhoto.jpeg -o MyPhoto.gif -xs 10 -ys 10

We can however order in any way we like, e.g.

ConvertJpegToGif -xs 10 -ys 10 -o MyPhoto.gif -i MyPhoto.jpeg

21.3 Getting the values

There are many string functions in C that we can use to extract parameters from the argv[] array (e.g. atof, atoi) however there is a very simple approach you may wish to use... **sscanf**

sscanf works just like **scanf** except if takes the input from an existing string rather than the keyboards, so if we knew argv[1] contained a integer value we wished to store in the variable age, we would simply write

sscanf(argv[1], "%d", &age)

sprintf

There is also an equivalent that 'prints' to a string - this is really useful for constructing (say) file names based on a loop counter.

The example below shows how we can create files file1.dat, file2.dat etc. within a loop

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int i;
7      char FileName[100];
8
9      for ( i = 1 ; i < 10 ; i++)
10     {
11         //'Print' text into string
12         sprintf(FileName , "file%d.dat" , i);
13
14         // Sidplay the name created
15         printf("Current file name: %s\n", FileName);
16     }
17
18     return 0;
19 }
```

Listing 21.1: sprintf example [c21\sscanf_example.c]

Chapter 22

In conclusion

This guide provides only an introduction to the C programming language,as you code more you will become aware of the many features of C - you will also develop your own programming style.

The key to being a good programmer however - and this applies no matter how complex the code you are writing, it to take time to consider what the code should do, write and test at a functional level (so knowing the individual parts that make up your code work as intended).

Other things to make you stand out as a good programmer

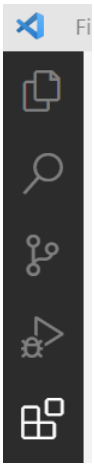
Comment you code as you go along - it is not something to do at the end

Assume users are idiots and code accordingly

Avoid the use of global variables

Appendix A: Setting up VSCode

Setting up VSCode for Compiling C Code



1. Download VSCode: https://code.visualstudio.com/download

2. Set up for compiling C:

- How to set up for C/C++ guide: https://code.visualstudio.com/docs/languages/cpp
- Install the C/C++ extension
    - Use the icon at the bottom of the toolbar in VSCode to select the Extensions view.
    - Search for 'c++'.
    - Select **Install.**

- Check if you have a C++ compiler installed
    - Open a new VSCode terminal window using Ctrl+Shift+`
    - Type `g++ --version` to check for the GCC compiler (use this for Windows or Linux)
    - Type `clang –version` to check for the Clang compiler in MacOS
- If no compiler is installed, install the GCC compiler (Windows):
    - Windows: Use MSYS2 https://www.msys2.org/ to install MinGW-x64
    - Follow the instructions on the webpage above carefully, **completing all 9 steps**.
- If step 8 above failed to show the gcc compiler it may be because it has not been added to the system PATH
    - Make sure that the compiler has been added to the PATH. Follow the instructions in step 7 of the 'Installing the MinGW-w64 toolchain' instructions here: https://code.visualstudio.com/docs/cpp/config-mingw#_prerequisites
- Check if the debugger was installed by typing `gdb –version` in the terminal window.
    - If the debugger was not installed use the MYSYS2 terminal to run the following command: `pacman -S mingw-w64-x86_64-gdb`
- Linux and MacOS should already have either gcc or Clang installed. If not, see the instructions in the setting up guide: https://code.visualstudio.com/docs/languages/cpp

- Install the C/C++ Runner extension:

This enables:

  - Building programs in both release and debug mode.
  - Building individual files in a folder (Ctrl + Shift + B)
  - Building and linking all the files in a folder (use the cog symbol in the bottom toolbar)

3. Install git - https://git-scm.com/downloads

- Add your name and email address to .gitconfig
  - Open either 'Git CMD' or 'Git Bash'
  - Set up your user name and email address by typing the following commands into the git terminal (substituting your own name and email:

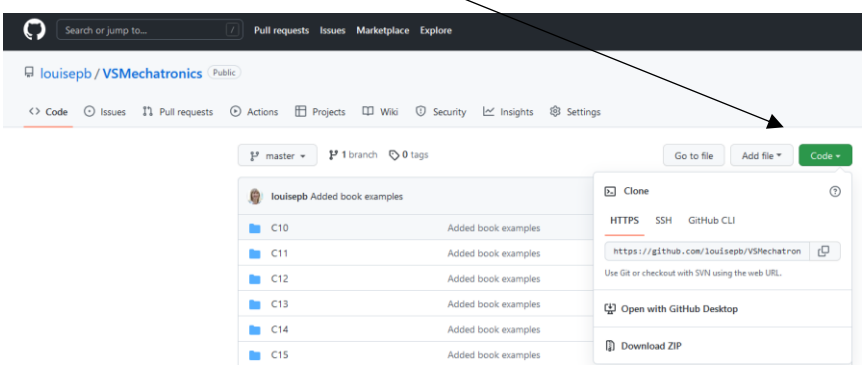
```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

- This should be sufficient to set up git for use with VSCode but further information can be found here: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

4. Download the code used during the course:

- Clone the VSMechatronics repository

  - Go to: https://github.com/louisepb/VSMechatronics.git
  - Select the green 'Code' button



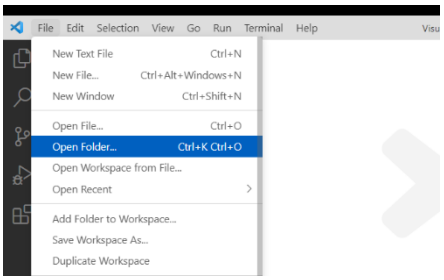
  - Use the 'Copy' button to copy the repository path

Appendix B: Starting VSCode

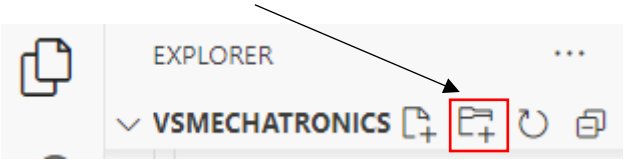
Quick Guide to Using VSCode

VSCode uses folders for storing code so think of a sensible structure for these to enable you to find your programs easily.

1. Open VSCode and then select **File->Open Folder.**

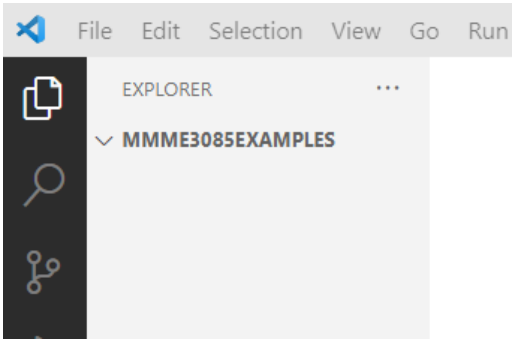


Either select an existing folder or create a new one using the New Folder icon.

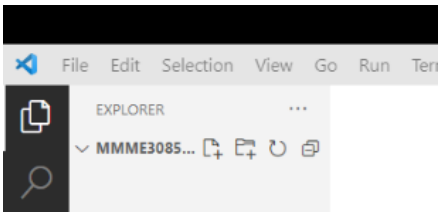


The selected folder will be shown in the window on the left. In this case a folder 'MMME3085Examples' has been selected.

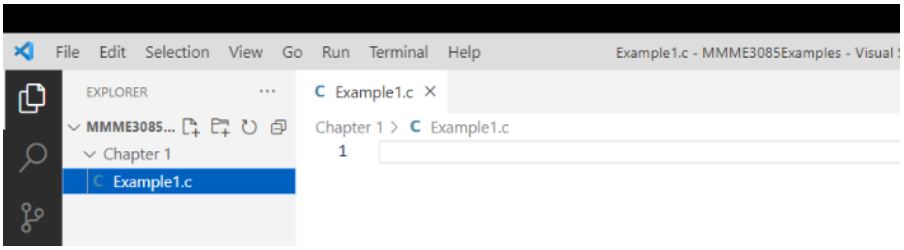
If there are any files in the folder these can be displayed using the down arrow at the left of the folder name.



2. Hover over the folder name with the cursor and icons will be displayed to open new files/folders. It might be a good idea to have a separate folder for each chapter in the exercises.



The icons can be used to create folders and new files within those folders. In this case a file called 'Example1.c' has been created. Make sure to use the '.c' file extension so that VSCode knows that it is a C file and knows how to compile accordingly.



3. Type the following into the Example1.c editor window:

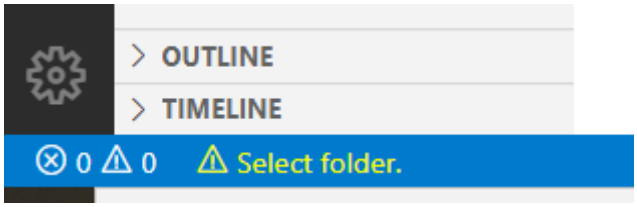
```
# include <stdio.h>

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

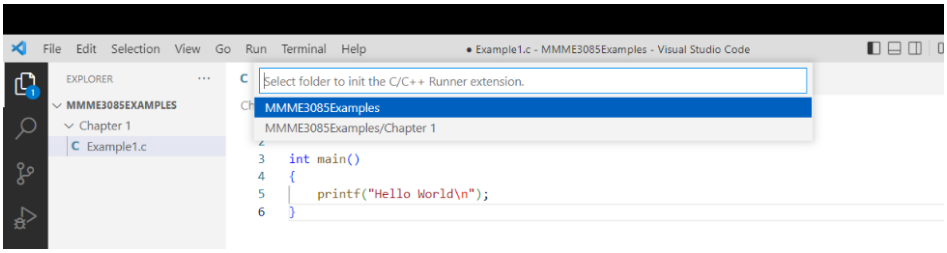
As you type you should see the intellisense giving suggestions of possible text.

4. The code now needs to be compiled in order to be able to run:

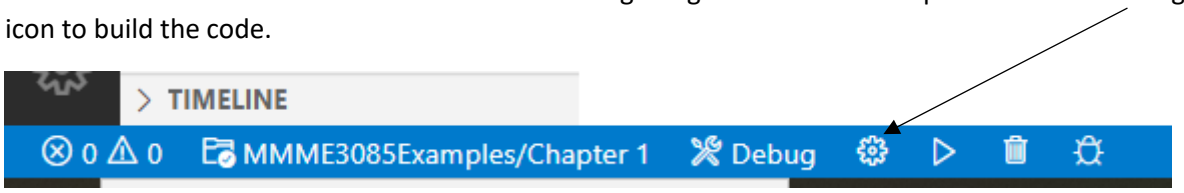
Select the 'Select folder' option from the toolbar at the bottom of the screen.



A dropdown box allows selection of the folder containing the code. In this case the 'Chapter 1' folder should be selected.



5. The toolbar at the bottom of the screen will change to give build and run options. Select the 'cog' icon to build the code.



You will see that a 'build' folder is created in the 'Chapter 1' folder. If you expand this you will see that this contains a '.o' file which is the compiled binary object file and the '.exe' file which is the final binary executable file created after linking the object file with any libraries (and other object files for more complex projects with more than one c file).

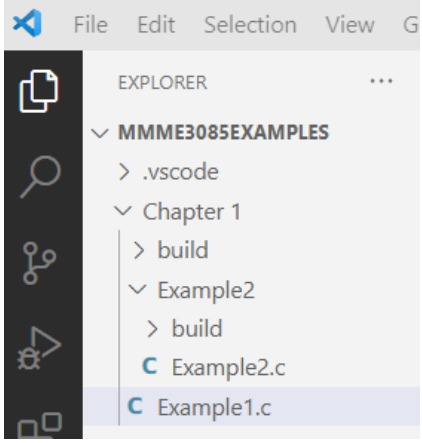
6. Select the arrow key to the right of the 'cog' key to run the program. You should see 'Hello World' appear in the terminal window at the bottom of th screen.

Congratulations! You have just created the classic C programming 'Hello World' program!

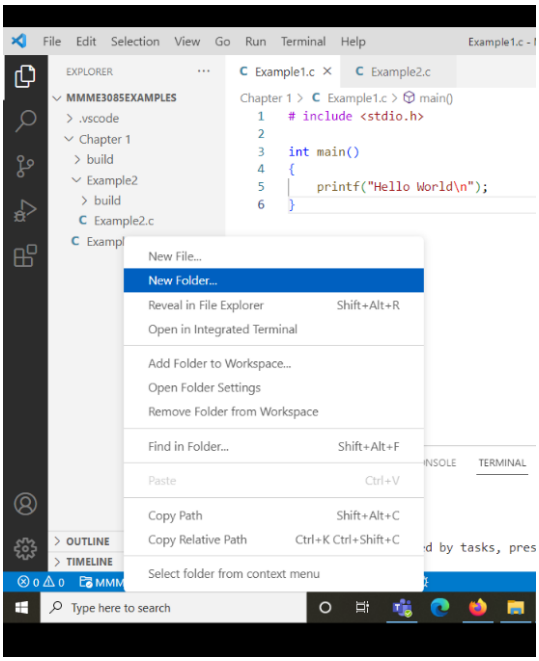
Adding a Second Program

A C program can only have one 'main()' function. To create a second program in VSCode it must be in a separate folder.

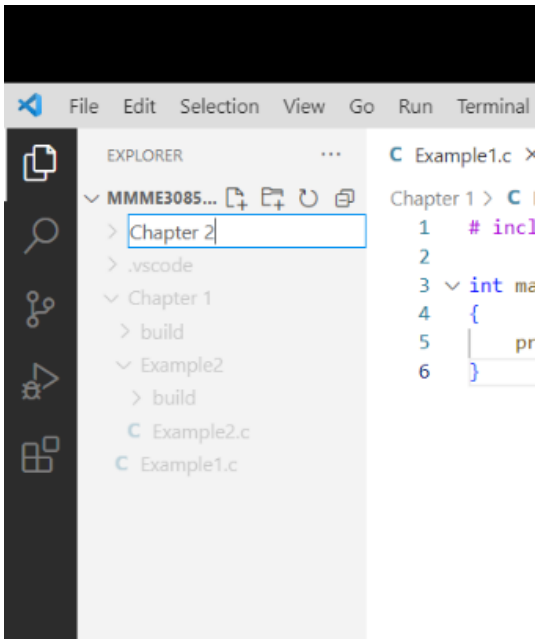
To create a subfolder, select the required folder and repeat the process from step 2. In the example on the right a subfolder 'Example2' has been created containing a file 'Example2.c'



To create a folder at the same level as the ‘Chapter 1’ folder, right click in the grey area below the folder structure and select ‘New folder’ from the drop down menu.



Type a new folder name in the space provided. A .c file can then be added and built as described earlier.



Note: To be able to successfully build and run the code in any folder there must only be one file with a main() function in that folder.

Appendix C: C Crib Sheet

H61 CAE– Revision Sheet

Variables

When using variables, you need to check the type you use is suitable to the value it will hold. For whole numbers you will generally use **int**, for real numbers use a **float**. A special case is a **string**, which is just an array of **char**'s

Type	Use For	Range (unsigned/signed)	Place Holder	Inputting	Outputting
char	Letters, small integers	0-255 / -127 -> 128	%c	x=getch()	putch(x)
int	Whole Numbers	0-65535 / -32768 -> 32767	%d	scanf("%d",&i)	printf ("%d",i)
float	Real values (non integer)	0->lots !	%f	scanf ("%f",&f)	printf("%f",f)
string: char s[**x**]	Whole words, sentences, filenames etc.	String of maximum length '**x**' Characters	%s	scanf("%s",s) gets(s)	printf("%s",s)

Looping

To count up/down use a **for** loop. To repeat while a condition remains true use a **while** or **do/while** loop

Counting up/down	While a condition is true, will not occur if condition is initially false	Happens at least once, repeats while condition remains true

for (initial_expression ; expression; inc/dec operation)
 	
for (i = 0 ; i < 100 ; i++)
{
code…..
}
 	
for (t = 9.0; t>8.0 ; t = t - 0.1)
{
code ….
}
	while (condition == true)
	{
	code ….
	}
		do
		{
		code ….
		}
		while (condition == true);

Decisions

For simple expressions use **if**. For a choice of options you can use **if/else if/else** or **switch** (the latter only works on single integer value comparisons and as such will not work over a range of values.

if (expression)
{ code… }
else if
{code… }
else
{code…}
 	
Expressions are made up using
== (equal to), != (not equal to)
>, >=, <, <=,
&& (and), || (or) , ! (not) etc.
	switch (statement)
	{
	case ***case1*** : code ; break;
	case ***case2*** : code ; break;
	default : code ; break;
	};
	
	'code' is executed from valid **case** until **break** is encountered. **default** case code is executed if no **case** applied.

Pointers

Point to the address in memory of a single variable or the element of an array. Pointers are always of the type of variable they point to and differ only in definition by the use of a * when defining. To obtain the address of a previously defined variable put a **&** in front of it. To access a variable via its pointer, use a * in front of the pointer pointing to it. Pointers are also used in conjunction with malloc/calloc (below)

Define variable & pointer	Assign pointer address of variable	Access variable via pointer
int a, *ptra;	ptra = &a;	*ptra = 7;

Arrays

These are analogous to matrices in mathematics. They can be of any dimension, but generally are 2D/3D. Arrays can be defined directly or by the use of malloc/calloc. The latter method has many advantages, one can ensure the array has been successfully created, create arrays at run time of a specified size and free up the memory when finished with. **Note:** Arrays are indexed starting a zero.

Simple method		Using malloc/calloc	
Define:		**Define:**	***type*** *tptr (***type*** is int, float, char etc)
int x[100]	: provides elements x[0] to x[99]	**Allocate**:	*tptr = (type *) calloc (no_items_required , sizeof (type));
float t[30]	: provides elements t[0] to t[29]		*tptr = (type *) malloc (no_items_required * sizeof (type));
Use:		**Check**:	if (tptr == NULL) -> memory allocation failed, react accordingly
x[1] = 200		**Use**:	x = tptr[10] , tptr[3] = 7 etc
t[28] = 0.23	etc.	**After use**:	free(tptr);

Functions

General definition: *return_type name (parameter list)*

name should relate to functions purpose

return_type is any variable type in C, or void if no value is to be returned

parameter_list - list of variable definitions separated by commas (or void)

For simple calculations where a single value is returned, one can define the function to have the required return_type and make use of the 'return' statement. To return/change more than one variable defined in main it becomes necessary to use pointers (see notes/lab work).

Files

Define a file pointer, assign to a file using fopen (fopen returns NULL in the case of an error), then use fprint, fscanf that operate the same as scanf & printf. To read to the end of a file make use of *feof,*

Define	Assign to file	Check opened K	Use	Close file

FILE *fptr	fptr = fopen (file_name, file_mode)
	
	file_mode : "r" - open for reading
	"w" – open for writing
		if (fptr == NULL)
		{
		error code..
		}
			Tex files: fprintf (fptr, "….." , …); cf printf
			fscanf (fptr, "….." , &…); cf scanf
			
			Binary files: fread(void *ptr, size_t s, size_t n, FILE *stream);
			fwrite(void *ptr, size_t s, size_t n, FILE *stream);
				fclose (fptr);

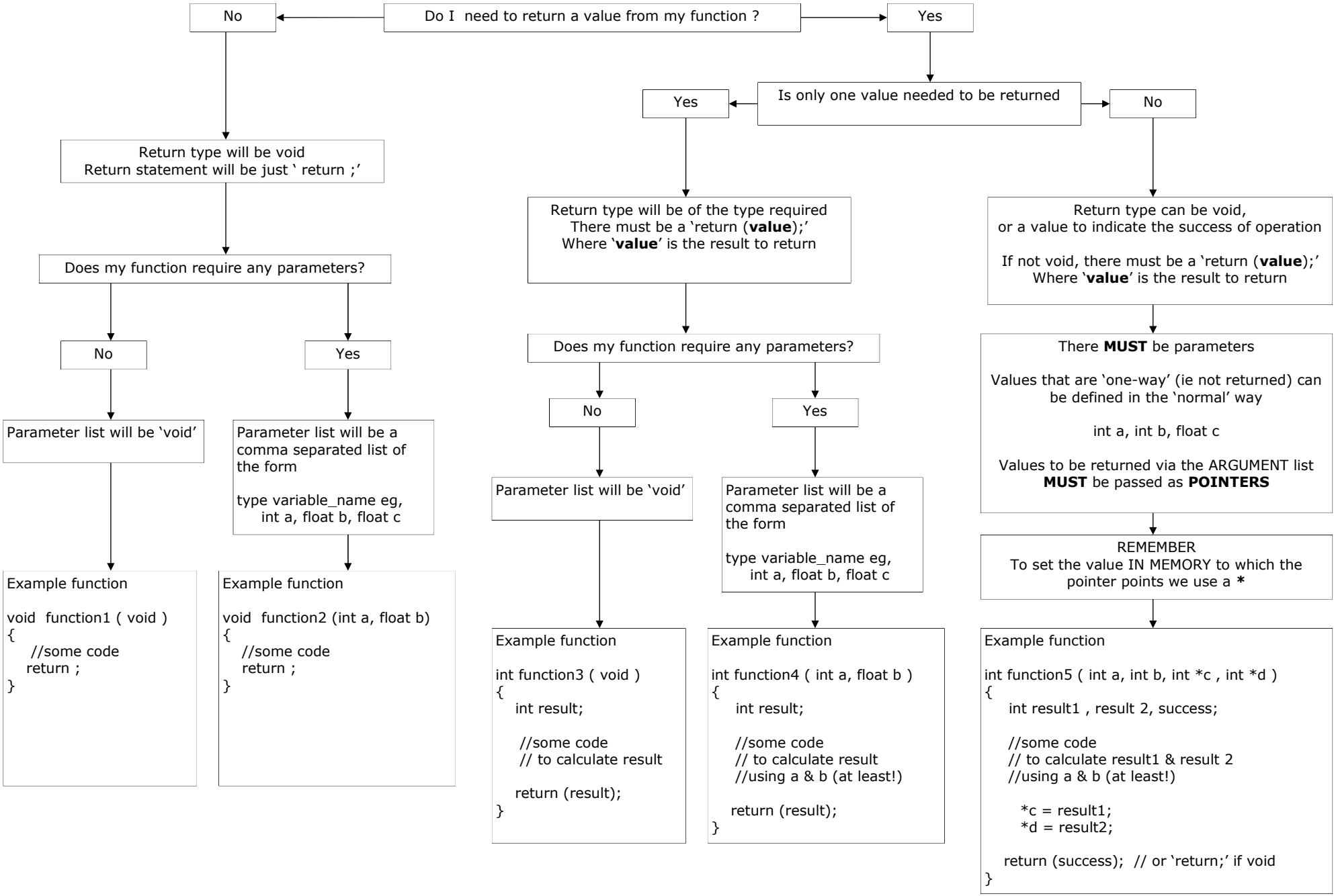
Structures

Allow a set of variables to be grouped together and treated as a single unit. Individual elements are accessed using '.element_name'. You can also define arrays of structures, in which case we use structure[array_index].element_name.

Define Structure	Create variable of type structure	Use structure & elements there within

struct MyStruct
{
int x,y,z; /* any valid type allowed */
}
	MyStruct MyRec1, MyRec2;
	MyStruct LotsRecs[100]
		MyRec1 = MyRec2;
		MyRec1.x = 23;
		MyRec1.x = MyRec2[y];
		LotsRecs[2].x = MyRec2.x

Appendix D: Function Flowchart


Do I need to return a value from my function ?
No
Yes
Return type will be void
Return statement will be just ' return ;'
Does my function require any parameters?
No
Yes
Parameter list will be 'void'
Parameter list will be a comma separated list of the form
type variable_name eg, int a, float b, float c
Example function
void function1 (void)
{
//some code
return ;
}
Example function
void function2 (int a, float b)
{
//some code
return ;
}
Is only one value needed to be returned
Yes
No
Return type will be of the type required
There must be a 'return (value);'
Where 'value' is the result to return
Does my function require any parameters?
No
Yes
Parameter list will be 'void'
Parameter list will be a comma separated list of the form
type variable_name eg, int a, float b, float c
Example function
int function3 (void)
{
int result;
//some code
// to calculate result
return (result);
}
Example function
int function4 (int a, float b)
{
int result;
//some code
// to calculate result
//using a & b (at least!)
return (result);
}
Return type can be void,
or a value to indicate the success of operation
If not void, there must be a 'return (value);'
Where 'value' is the result to return
There MUST be parameters
Values that are 'one-way' (ie not returned) can be defined in the 'normal' way
int a, int b, float c
Values to be returned via the ARGUMENT list MUST be passed as POINTERS
REMEMBER
To set the value IN MEMORY to which the pointer points we use a *
Example function
int function5 (int a, int b, int *c , int *d)
{
int result1 , result 2, success;
//some code
// to calculate result1 & result 2
//using a & b (at least!)
*c = result1;
*d = result2;
return (success); // or 'return;' if void
}