First Year Computing Course:
An introduction to procedural programming

Dr G. A. Wynn, Dr R. G. West, Prof. R. Willingale

Department of Physics & Astronomy, University of Leicester

September 2012

Introduction

Computers are used to perform a bewildering array of tasks. They have become an essential part of life, and some set of computing skills are required in almost all modern vocations. As a physics undergraduate you will gain experience of a wide range of these skills. You will use computers for a number of tasks;

- experiment control
- data collection
- data analysis
- image processing
- computational physics
- word processing

All of these tasks require the use of simple keyboard skills, text editing, simple operating system commands and software that other people have written. The next section provides a brief introduction to these simple skills. In addition, as a physicist, you will also want to programme computers yourself to solve problems and perform tasks which you have devised. To do this you will need to learn how to tell a computer what you want it to do. This workshop is your first step towards becoming a competent computer programmer as well as a skilled computer user.

Operating Systems and Programming Languages

Computers are quite stupid. They will do *exactly* what you tell them do to, so you will find that you have to tell them what to do very carefully. The first problem is how do we communicate with the computer at all? We do this through the operating system, which provides a way for the user to tell the computer's engine (the central processing unit or CPU) you want it to perform a certain task. There are many types of computer and operating systems and you will meet several different systems as you progress through your degree course. You may already be familiar with one of the most popular operating systems: Microsoft Windows, which runs on many personal computers. We have attempted to design the first, second and third/fourth year computing workshops in a coherent way so that you gain a depth of experience in a system environment which is close to that used by professional physicists. In this workshop you will be using the same hardware and software as research workers in the University.

The computer service you will use is called SPECTRE and runs a version of Linux which is a Unix-like operating system. UNIX is a very widespread operating system which has been adopted by many different computer manufacturers, so when you are familiar with Linux you will be able to use a large number of different machines.

It is important at this stage to understand the difference between the operating system and a programming language:

- An operating system is a computer program which has been written to allow you to tell the computer how to perform any number of tasks, which may include for example looking the contents of a directory or file and running software written by yourself or other people. You will tell the computer to perform these particular tasks during the course of this workshop.
- A high-level programming language (such as 'C') consists of a set of instructions which allow you tell the CPU how to perform a particular task (the set of instructions as a whole is called a computer program). Usually the task is something that is not catered for by the operating system or any third party software. In fact, when you ask the operating system to do something it will often run a computer program similar to those you will encounter in this workshop.

The rest of this workshop is intended to give you enough information on the Linux operating system to write, and run your own computer programs in the 'C' programming language. As you progress through the workshop ***please keep notes on what you do***, and keep listings of your programs.

SPECTRE: getting started

The first thing to do is to access the SPECTRE computer system. In order to log to the machine you will need a user name and password for the UoL IT service. You should have already received your account details at the beginning of term.

The X-Windows Client NX

You gain access to the SPECTRE machines using a PC running Windows. The program you must run on the Windows machine is an X-Terminal client called NX.

- Log on to Windows (using your UoL IT username and password)
- Install NX (only needed once)
    - Programs→Program Installer →NX Client (click)
- Start up NX to log in to SPECTRE
    - Programs→NX Client (click) SPECTRE (click)
- Login to SPECTRE using your username and password
- Start a Terminal Command Line window
    - Applications->Accessories-> Terminal (click)

You must make sure you type your username and password in the correct case (either capitals or lower case) as UNIX/Linux operating systems are case-sensitive. Your password is known only to you and will not appear on the screen as you type.

When your terminal window starts up you should get the system prompt which ends in a dollar sign something like [zrw@spectre02 ~]$. This tells you who you are, which machine you are using and the current files directory you are in and it is the operating system soliciting for commands from you. Anything you type and enter (with the RETURN key) at this prompt will instruct the computer to do something.

If you make an error in typing a command, or want to repeat a previous command, you can use the *command-line recall* feature. Pressing CTRL-P (press and hold CTRL, press P, then release CTRL) will recall previous lines that you have typed. Using CTRL-B and CTRL-F you can move the cursor backward and forwards along the line to edit it.

Once you have finished using the Terminal Window you may logoff by typing exit (and hitting the 'RETURN' key)

```
Exit
```

You must log out of SPECTRE session entirely using

- System→Log Out(click)

******* Never leave a computer logged on to an account unattended!! *******

The EMACS and NEDIT Editors

Information is stored on a computer in files. A file may contain many things, including plain text, a computer program, an image or a data set. In this workshop you will need to create and manipulate files. One way to create or edit a text file is to use software known as an *editor*. The editors we will adopt for this workshop are called ***emacs*** **(***which is more powerful***)** *and* ***nedit*** **(***which is simpler and more like using word***)**. To invoke the editor you need to enter the commands

```
emacs filename &
```

or

```
nedit filename &
```

where `filename` is the name of the file you wish to edit. Invoke the emacs editor to create a new file by entering the command

```
emacs firstfile.txt &
```

If you have entered the command correctly a new window should open up on your terminal. This window shows the contents of the file `firstfile.txt`, which is empty at present. Type some text into the window and then save the contents of the file and exit emacs by holding down the control key while typing 'X' followed by 'C'. Answer any questions which appear in the bottom line of the emacs window. You have now created your first file. A listing of all of the files in the current directory (find out about directories in the next section) may be obtained by typing `ls`.

Entering this command should prompt the computer to list the name of your new file:

```
firstfile.txt
```

You may examine its contents using the command `more`. Entering the command

```
more firstfile.txt
```

into the computer should prompt it to reveal the text you typed into the emacs window.

In this workshop you will be asked to create files containing computer programs written in C. When you are required to do this give your file a sensible name followed by the extension .c, an example would be

```
emacs program1.c
```

Some of the more useful emacs commands are listed below. You will have noticed that an ampersand (&) has been added at the end of the emacs command lines above. This runs emacs as a background job so that you can use the mouse to toggle between the emacs window and the original xterm window. When you have made an edit you can save the buffer using c-x c-s (see below) and then move to the xterm window WITHOUT killing emacs. Making edits, re-compiling and re-running your program is faster if you do this and you can still see your source code in the emacs window when you are running your program which can be useful.

EMACS Crib Sheet

Commands using "Control" ("C-" means hold down the Control, or CTRL, key at the same time as hitting the appropriate alpha or numeric key).

`C-x C-f`	Read a file (type the filename in the mini-window)
`C-x C-s`	Save the file in the specified filename (or type a name). If it's okay to save, just hit "Return"
`C-x k`	Remove a buffer from emacs
`C-x C-c`	Kill emacs: if you have unsaved files, it will ask if you want to save them
`C-x 1`	If the emacs screen is split, revert to a single file display
`C-x 2`	Split the screen to display 2 files in emacs
`C-g`	Kill the current command in the mini-window
`C-k`	Delete the current line
`C-d`	Delete the next character
`C-v`	Scroll down one screen length

Note : The mini-window is the single line at the bottom of the emacs window, which shows the emacs commands, and their results.

Files and directories

Once created, files can be stored in directories. It is good practice to store associated files together in directories. To make a directory called `Cworkshop` enter the command

```
mkdir Cworkshop
```

Now entering `ls` will prompt the computer to return

```
Cworkshop firstfile.txt
```

You can move the file firstfile.txt into the directory `Cworkshop` by entering the command

```
mv firstfile.txt Cworkshop
```

Entering `ls` should now only list the directory `Cworkshop`. You may list the contents of the directory Cworkshop by entering

```
ls Cworkshop
```

You can change your working directory by typing

```
cd Cworkshop
```

all of the files you now create will be stored in this directory. If you want to move up a level in the directory hierarchy, use

```
cd ..
```

Files can be deleted using the `rm` command, eg.

```
rm firstfile.txt
```

What is a computer program?

The fundamental purpose of a computer program is to perform some action or solve some problem. Often this action or problem will be something you as a user might express in English in a very concise way, for example "show me my e-mail", or "show me the BBC News web page", or "play Quake 3".

Despite what science fiction movies will have you believe, modern computers are actually very stupid. A typical modern CPU (Central Processor Unit, the heart of a computer) is capable of performing a limited range of very simple operations (typically twenty or so). These operations are of the form "get this number from here", "add this number to that number", and "put that number over there". However, it is capable of carrying these simple instructions out very quickly (in excess of a billion instructions per second).

Clearly there is an enormous gap between the language that a human will typically use to express a problem, and those used by the CPU. Programming languages are designed to help bridge this gap. There is a wide range of programming languages in use today, but they all serve the same basic purpose – they try to make it easier for a programmer to express a human problem in terms that a computer is capable of understanding.

A computer program is essentially just an ordered list of simple instructions. The challenge in writing a computer program is breaking down the problem ("play Quake 3") into a correct sequence of simple instructions.

Computer programs can be very large indeed; most computer games are over a million lines of code, and the Microsoft Windows 2000 operating system consists of over 35 million lines of code. When translated into the instructions understood by the CPU this number is typically increased by somewhere between ten- and a hundred-fold.

Compiling and running a C program

As we explained in the previous section most programming languages you will encounter (so called *high-level languages*) have been developed to be easily understood by humans. Before a program can be run it must be translated into the simple instructions which are understood by the CPU. This translation step is called *compilation*, and is achieved by a *compiler*.

Using the compiler is quite simple. Say you have a C program in a file called `prog1.c`. To compile it you should use the following command-line:

```
$ gcc prog1.c –lm –o prog1
```

This will create a new file that contains the translated version of the program, expressed as instructions which can be understood by the CPU. We call such a file an *executable*, as it is capable of being executed by the CPU (which your original source code was not). The `-o` tells the compiler what filename to give the executable file (`prog1` in the example). The `-lm` switch allows us to use mathematical functions in our programs.

To run this newly created program, you just use the filename of the executable (adding `./` tells the computer to look in the current directory for the file to run):

```
$ ./prog1
```

Program 1: Variables, basic arithmetic, and printing numbers

Here is a simple 'C' program:

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

```
int main()
{
     /* Declare variables */
     float a, b, sum;

     /* Assign values to variables */
     a = 10.0;
     b = 2.0;

     /* Calculate the sum, print it out */
     sum = a + b;
     printf("The sum is %f\n", sum);
}
```

***Exercise 1**: Type in this program, compile it and run it. You will need to:*

- *create a new file in which you can enter the text (*`emacs prog1.c`*)*
- *compile the program (*`gcc prog1.c -lm -o prog1`*)*
- *run the program* (`./prog1`*)*

The program introduces several very important concepts in computer programming. Ignore the first six lines for now (see the later section on" Symbolic constants and the pre-processor"), just concentrate on the lines between the curly brackets (`{`, and `}`). The program is basically a short list of instructions (called *statements* in C). In the C language each statement is separated from the next by a semi-colon (`;`). **Be very careful to check that you place semi-colons where they are needed, missing one out will generate a lot of error messages.**

The program also contains comments, which are enclosed (delimited) by `/*` and `*/`. The comments are ignored by the compiler, so you can put whatever you like in the comments (except other comments, ie. comments cannot be nested). The point of a comment is to aid readability, to explain to another human reading the program what the program is doing (or supposed to do) at a given point. Learning to make liberal use of comments is an important part of learning to program a computer. A program is as good as useless if no-one can tell what it does!

The first thing the program does is to declare three *variables*. A variable is a named entity that holds a value. The value of a variable can be updated as a program runs (hence the name). Every variable must have a *type*. The C language supports both numeric and non-numeric types; we will leave non-numeric types until later. Computers use two different types of numbers: integers and real numbers (commonly called floating-point numbers). The line

```
float a, b, sum;
```

declares three variables named `a`, `b` and `sum`, and states that they contain floating-point (real) values. If the word `float` was replaced by `int` then the variables would be declared as integers.

The next action the program takes is to assign values to variables. This is called *defining* the variable value. Before these lines all three variables will contain an *undefined value*. It is a common mistake to declare a variable but forget to give it an initial value. Most modern compilers will warn you if you declare a variable and never use it, or use it before you assign it a value (though it is very bad practice to rely on this).

After the variables `a` and `b` are assigned values they can be used in an arithmetic expression. The assignment bears a passing resemblance to an equation in algebra (though *it is not* an algebraic equation):

```
sum = a + b;
```

What this line does is add the values held in variables `a` and `b`, and assign this value to the variable named `sum`. (A CPU would execute this using simple instructions like "get this number from there", "add this number to that number" and "put that number there").

The last thing the program does is print out the result. After all, a program which keeps its answers to itself is pretty useless!

***Exercise 2**: Alter the program (`prog1.c`) to print the sum of two different numbers.*

Variable names, and how to choose them

Variables in C must be given a name. The *variable nam*e can be made up from upper-case or lower-case letters (A—Z or a—z), numeric characters (0—9) or an underscore ('_'). There is one important restriction though: **a variable name cannot start with a numeric character.** Variable names are case-sensitive in C, in other words "`value`" and "`Value`" are different variables. It is a **bad idea** to define variables with names that are distinguished only by their case, as this makes code difficult to interpret, and can lead to mistakes that are very hard to track down.

Most implementations of C allow variables to be given long names (at least 32 characters). You should choose names for your variables that are reasonably descriptive of what the variable represents (to help other programmers read your code, for example `nvalues`) without being unnecessarily long-winded or difficult to type (eg. `number_of_values_in_the_file_that_I_am_reading_on_Tuesday`).

Numeric data types

Numeric variables in computer programs are not capable of representing every possible real or integer number; they have limited precision and can only represent a limited range of numbers. There is a trade-off between precision/range and the amount of storage space required to represent the number held within the variable. The table below lists some real and integer data types that you may find useful. A byte consists of 8 bits.

Type	Range	Precision	Storage size
`int`	-2147438646 to 2147483647	Integers only	4 bytes

The `float` data type is often referred to as *single-precision*, and the `double` type as *double-precision*. Historically single-precision mathematical operations were faster than double-precision, however with modern CPUs this is generally no longer the case.

Printing output and formatting numbers

The `printf` statement is used to print output to the screen. `printf` is an example of a *function* in C. A function is a separate piece of code that you can call from your program to perform a specific task. When calling a function you can supply *arguments*, which are variables or values which the function will need to perform the task you want it to. These arguments follow the function name and are enclosed in brackets. Functions are a very important part of programming in C; a large number of powerful functions are provided for you to use, and you can even write your own functions. We will come back to functions in more detail later on.

Fortunately you do not need to know how the `printf` function works internally, you only need to know what arguments to give it to print the output you want. This is part of the

purpose of a function; your program does not need to concern itself with the mechanics of formatting numbers and printing them out; this has all been done for you.

The `printf` function accepts one or more arguments. The first argument is a *format specifier*, which is basically a template for what you want printed out. Take the example in the previous program:

```
printf("The sum is %f\n", sum);
```

In this case the format specifier is the string `"The sum is %f\n"`. What the `printf` function does is to work through the format specifier string and replace the format codes (which look like '`%f`') with the formatted representation of the values of the subsequent arguments in order.

The format code `%f` is used to format real numbers, the format code `%d` is used for integer values. **Be very careful to ensure that the format codes are correct for the data types of the arguments, otherwise you will quite probably get garbage!** The '`\n`' character sequence stands for "newline", and asks `printf` to start a new line on the screen.

The example below shows both integer and real numbers in the same line of output:

```
int first;
float second;

first = 10;
second = 11.0;

printf("The value of first is %d, the value of second is %f\n",
             first, second);
```

would print

```
The value of first is 10, the value of second is 11.0
```

***Exercise 3**: Alter* `prog1.c` *to print both the sum and the product of the two values. The asterisk symbol ('*') is used to represent multiplication.*

Arithmetic expressions

The C language supports the following operators for use in arithmetic expressions:

+	Addition
–	Subtraction
*	Multiplication

In complicated arithmetic expressions the compiler will use rules of *operator precedence* to work out the order in which the operations should be performed. Operator precedence can be tricky to get to grips with (see the appendix for the full precedence rules).

For now it is enough to know that multiplication and division have higher precedence than addition and subtraction, for example:

```
a * b + c * d
```

will evaluate `a * b`, then `c * d`, then add together the two intermediate results. If you are in doubt how an expression might be interpreted by the compiler, it is best to use parentheses (brackets) to make sure it understands what you really mean.

Parentheses can be used to dictate the order in which parts of an expression are evaluated, for example:

```
a * ( b + c ) * d
```

will evaluate `b + c`, then multiply this intermediate result by `a` and `d`.

It is possible to mix numerical types in an expression, for example to multiply an integer (`int`) by a single-precision floating-point number (`float`). When such an expression is evaluated the values of the variables will be automatically converted to the same type as the highest precision variable used in the expression (for example `int * float` becomes a `float`, `int * float * double` becomes a `double`).

It is also legal to assign values of one numeric type to variables of another type, however in some cases this will lead to loss of precision, so care should be taken when doing this. When real numbers are assigned to integer variables, the values are truncated (*not* rounded, ie. 67.9 becomes 67, *not* 68).

***Exercise 4**: Modify* `prog1.c` *to divide two integers and assign the result to an integer variable. What can you say about the results? What happens if you assign it to a* `float`*?*

Program 2: Repeating instructions

How might you modify Program 1 to calculate and print, say, the factorial of the variable `a`? Remember that a factorial is defined as:

$$a! = 1 \times 2 \times 3 \times 4 \times \ldots \times a$$

so basically you must multiply `a` by all integers less than `a`. An obvious way to do this might be to modify the program to explicitly perform the multiplications required as follows:

```
fact = 1.0;
fact = fact * 2.0;
fact = fact * 3.0;
…
fact = fact * a;
```

however this is not very satisfactory because firstly it involves a lot of typing, and secondly that the program is difficult to modify if you want to calculate the factorial of a different number.

A more elegant (and the correct) solution is a *loop*. What a loop does in a computer program is to execute a bunch of instructions more than once. A loop will continue to execute the enclosed instructions until some condition (defined by the programmer) is satisfied. The following program will calculate the factorial of a number:

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

int main()
{
    int i, num;
    float fact;

    /* What number do we want to calculate the factorial of? */
    num = 10;
```

```
    /* Initialise the factorial */
    fact = 1.0;

    /* Count from 2 to num, multiply fact by counter each time */
    i = 2;
    while ( i <= num ) {
       fact = fact * i;
       i = i + 1;
    }

    /* Print the result */
    printf("The factorial of %d is %f\n", num, fact);
}
```

***Exercise 5**: Type in this program (call it* `prog2.c`*), compile it and run it.*

This program makes use of a `while`-loop. The basic syntax of a while loop is:

```
while ( condition )
    statement
```

in other words it repeatedly executes `statement` *while* `condition` is satisfied. The program uses the variable `i` as a *counter*, which starts with a value 2 and is repeatedly incremented. Each time it is incremented a second variable, `fact`, is multiplied by the counter. This continues while the value of `i` is less-than-or-equal-to the value which we are trying to calculate the factorial of (`num`).

The while loop in this program is an example of what is known as a *definite loop*. A definite loop executes a fixed number of iterations.

Logical expressions and relational operators

In an earlier section we covered *arithmetic expressions* which are used to calculate numerical values (which are typically then assigned to a variable or passed as an argument to a function).

The C language also supports a different type of expression, the *logical expression*, which can be used to test whether a certain condition is true.

In the example program above the `while` loop has an associated logical expression

```
while ( i <= num )
```

In this case the logical expression `i <= num` evaluates to "true" so long as `i` is less-than-or-equal-to `num`.

The following relational operators are available in C:

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Note that is a *very* common mistake to mix up the equality test operator (==) with the assignment operator (=). It is a quirk in the C language that the assignment operator is legal in all the contexts in which the equality test is valid, however they do very different things. Most modern compilers will attempt to spot this mistake and issue a warning, but this cannot be a substitute for due care and attention when constructing logical expressions!

***Exercise 6**: Alter* `prog2.c` *so that the counter counts from num down to 1, rather than the other way.*

***Exercise 7**: A single precision floating-point variable can only represent values up to about 10^{38}. What is the largest value of* `num` *for which this program can calculate the factorial? Alter the program to use double precision floating-point, which can represent values up to 10^{308}. What is the largest value of* `num` *for which the factorial can be calculated now?*

C also provides a means by which conditions can be chained together. Suppose you want to test whether the variable `a` is within the range 1.0 to 10.0. This would be written as:

```
a >= 1.0 && a < 10.0
```

The `&&` operator (AND) is a *boolean operator* which links the two sub-expressions together; it is only "true" if the first sub-expression is "true" AND the second sub-expression is "true".

Now suppose you wanted to test whether the variable `a` is outside that range. There are actually two ways to do this. Firstly you can use the OR operator (`||`):

```
a < 1.0 || a > 10.0
```

The `||` operator (OR) is "true" if the first sub-expression is "true" OR the second sub-expression is "true".

The alternative way to perform the test is to check whether the value is within the specified range, and invert the result, using the ! operator (NOT). The value must be outside the range if the value is NOT inside the range:

```
! ( a >= 1.0 && a <= 10.0 )
```

Note that in this example parentheses are used to dictate the order of precedence as they were in arithmetic expressions. Logical expressions are subject to rules of operator precedence in much the same way as arithmetic expressions are. The basic rules are that the operators <, <=, ==, >= and > have higher precedence than `&&` or `||`. The NOT operator (`!`) has higher precedence than all of the other relational operators, hence the need for the parentheses in the example above, which would otherwise be interpreted as if it were written:

```
( ! a >= 1.0 ) && ( a <= 10.0 )
```

The `for`-loop

The loop in the example Program 2:

```
i = 1;                          /* Initialise counter variable */
while ( i <= num ) {            /* Test counter variable */
     fact = fact * i;
     i = i + 1;                 /* Increment counter variable */
}
```

performs three important operations (highlighted in bold text). Firstly it sets a counter variable (in this case `i`) to an initial value (1), secondly it tests that the counter variable is still within the required range, and thirdly it increments the counter variable.

This form of loop, the definite loop, is so common in C that a special shorthand statement has been devised for it, which is the `for` statement. The `for` statement encapsulates the three

key operations into one statement, which helps readability. The code above can be replaced by the following:

```
for ( i=1; i <= num; i = i + 1 ) {
     fact = fact * i;
}
```

which is much less long-winded, and expresses all you need to know about where the loop starts, stops, and how the counter behaves in between, in one single statement. In general you should always use a `for` statement to control definite loops.

Exercise 8: *Modify* `prog2.c` *to use a* `for` *statement, rather than a* `while` *statement.*

Compound statements

You have probably noticed that the example programs presented so far contain quite a few curly brackets ({ and }). Curly brackets are used in C to delimit (enclose) what is called a *compound statement*.

A compound statement is quite simply a block of code made up from one or more statements. A compound statement can be used pretty much anywhere a single statement can be used.

When we introduced the `while` loop we said that the syntax was:

```
while ( condition )
     statement
```

You can see that in the example program `statement` is actually a compound statement consisting of the following two statements:

```
{
     fact = fact * i;
     i = i + 1;
}
```

Once we replaced the `while` loop with a `for` loop there is actually only one statement executed by the loop, so strictly speaking we can drop the curly brackets:

```
for ( i=1; i <= num; i = i + 1 )
     fact = fact * i;
```

You may see this often if you are reading a C program written by an experienced programmer. It can be quite confusing at first working out where curly brackets are needed and where they aren't. Just remember that a compound statement is intended to make lots of statements look like a single one. **Be very careful that every curly bracket you use to open a compound statement has a companion bracket that closes that compound statement. If you don't you will almost certainly get an error message and your program won't compile.**

Exercise 9*: Bearing in mind what you have learned about compound statements, what is wrong with writing the loop above as follows:*

```
while ( i <= num )
     fact = fact * i;
     i = i + 1;
```

This program is syntactically correct, and will compile without an error. What do you think will happen when it is run? Change `prog2.c` *and find out! You may want to save a copy of the version of the working program in a new file. Use the IRIX* `cp` *command to do this;*

```
cp prog2.c prog2saved.c
```

Program 3: Entering numbers into a program

So far the example programs have been performing arithmetic operations on values of variables which are hard-coded into the program, for example:

```
a = 10.0;
b = 2.0;
sum = a * b;
```

This approach is not very useful if you want to write flexible programs. You don't want to have to edit the program and recompile it every time you want to calculate the product of a different pair of numbers. What is needed is some way to get numbers into a program, say by typing them at the keyboard when the program runs, or by reading them from a text file.

Fortunately this is quite straightforward to do. Earlier we saw how the standard function `printf` was used to format text and numbers and print them to the screen. There is another standard function, called `scanf`, which performs the opposite operation, namely reading values from the input and assigning these values to variables.

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

int main()
{
     float a, b, sum;

     /* Read two numbers */
     printf("Enter two real numbers: ");
     scanf("%f %f", &a, &b);

     /* Calculate the sum, print it out */
     sum = a + b;
     printf("The sum is %f\n", sum);
}
```

You can see that `scanf` looks very similar to `printf` in this example. It reads two real numbers from the keyboard, and assigns the values to two variables `a` and `b`. It uses the same format specifier codes (`%f` to read a real number, `%d` to read an integer) as `printf`. The only difference is that the second and third arguments, which are the variables intended to receive the two values are prefixed by an ampersand (`&`) in the argument list.

The reason for the ampersand is that it asks the C compiler to ensure that the variables are passed to the function in such a manner that the `scanf` function can actually write values to the variables. It does this by passing the *address* of the variable, rather than the value of the *variable*. This allows the function being called (`scanf`) to write the value to the correct location in the computers' memory. You will learn more about this subject (*pointers*) in the Second Year course.

***Exercise 10**: Type this program into a new file (call it* `prog3.c`*), compile it and run it.*

Program 4: Subscripted variables (arrays)

So far the variables we have been dealing with have held a single value (a scalar). For some problems you might want to collect together a number of related numeric values, for example a list of measurements of a length, or a temperature, or similar.

The naïve approach is to declare N distinct variables, each with a different name to hold each of the distinct values you are dealing with, for example:

```
float value1, value2, value3, value4;
value1=10.0;
value2=12.0;
value3=15.0;
value4=17.0;
```

and then to explicitly work with these variables. Clearly this is very cumbersome and inflexible. The program will have to be altered if the number of values changes, it is very difficult to perform mathematical operations on these values (for example subtracting a constant from all the values, or similar). Imagine how long a program would be if it had to deal with hundreds, thousands or millions of distinct values!

Fortunately there is a simple solution offered by almost all programming languages, and C is no exception. It is also possible to define a single variable that represents a collection (or array) of values. An array variable is declared as follows:

```
float value[100];
```

This code fragment declares an array of 100 distinct single-precision real values, and associates them all with the variable name "`value`".

The square bracket notation is also used to access the individual elements of the array, for example `value[0]` accesses the first element of the array, `value[1]` the second, and `value[99]` the last. Array indices are always integers. **Note that the first element of the array is numbered zero, and therefore that** `value[100]` **is not a valid element of this array. This is a very common source of confusion to programmers new to C.** Accessing non-existent elements of an array can cause all sorts of problems. At best you will get the wrong answer, at worst your program will crash with a *segmentation fault* or *bus error*.

The advantages of arrays are many-fold. For a start it is possible to use another variable as an array index (ie. within the square brackets), which means that operations can easily be carried out on all values in an array simply by using a loop:

```
int i;
float value[100];

for ( i = 0; i < 100; i = i + 1 )
     value[i] = value[i] + 10.0;
```

This example will add 10.0 to the value of every element of the array; much easier to write and understand than had the values all been held in distinct variables. We can also perform some very important operations on arrays that don't really make sense for a scalar variable (for example sorting that values into some order).

The following program will read ten numbers into an array, then calculate the mean and standard deviation of the values:

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

int main()
{
     int i;
     float value[10], sum, mean, stddev;

     /* Read ten values from the keyboard into the array */
     for ( i = 0; i < 10; i = i + 1 ) {
```

```
        printf("Enter value %d: ", i);
        scanf("%f", &value[i]);
    }

    /* Calculate the mean */
    sum = 0.0;
    for ( i = 0; i < 10; i = i + 1 )
        sum = sum + value[i];
    mean = sum / 10.0;
    printf("Mean value is %f\n", mean);

    /* Calculate the standard deviation */
    sum = 0.0;
    for ( i = 0; i < 10; i = i + 1 )
        sum = sum + powf(value[i] - mean, 2.0);
    stddev = sqrt(sum) / 10;
    printf("Standard deviation is %f\n", stddev);
}
```

Note that the `powf()` function calculates the first argument raised to the power of the second, so `powf(2.0, 3.0)` would return 8.0 (which is 2^3). The `sqrt()` function calculates the square-root of the argument.

***Exercise 11**: Type this program into a new file (call it `prog4.c`), compile it and run it.*

Return values from functions

You have seen how, when calling a function, you can specify a number of arguments (values which are to be used by that function to perform the operation you are requesting of it).

Once these operations have been performed, the function can return a value back to the calling program. This value, called the *return value*, can be assigned to a variable, or used in an expression. In the example above we are using two functions, `sqrt()` and `powf()` in this way, embedding them directly into numeric expressions.

Hopefully it is obvious why `powf()`, `sqrt()` or other mathematical functions should return the value they calculate (they would be pretty useless otherwise!), however you should note that a large fraction of non-mathematical functions also return a value. Often this return value is used to communicate back the success (or otherwise) of the execution of the function. Even `printf()` returns a value, which indicates the number of characters actually printed, as does the `main()` function, which is the main body of the program itself.

In C it is legal to call a function and ignore its return value; this is we have been doing when calling `printf()` in the previous example programs. If you don't assign the value to a variable, or use it in an expression, the return value is simply discarded.

Program 5: Input from a file

Reading data from the keyboard is very useful, however if you have a large amount of data, or need to run a program on the same data time and again, it is not very sensible to repeatedly type the numbers into the program by hand. The solution is to type numbers into a separate file, then read them from that file as if they had been from the keyboard.

The following program will read numbers from a file and calculate their mean and standard deviation:

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

```
#define MAXVALS 100

int main()
{
    float value[MAXVALS], sum, mean, stddev;
    int i, count;
    FILE *fh;

    /* Open the data file */
    fh = fopen("file.dat", "r");
    if ( fh == NULL ) {
       printf("Cannot open data file\n");
       exit(EXIT_FAILURE);
    }

    /* Initialise counter */
    count = 0;

    /* Read all the values in the file */
    while ( ! feof(fh) && count < MAXVALS ) {
       if ( fscanf(fh, "%f", &value[count]) > 0 ) count = count + 1;
    }

    /* Close the file */
    fclose(fh);

    /* Check the file contains data */
    if ( count == 0 ) {
       printf("Data file contains no values\n");
       exit(EXIT_FAILURE);
    } else {
       printf("Read %d values from file\n", count);
    }

    /* Calculate the mean */
    sum = 0.0;
    for ( i = 0; i < count; i = i + 1 )
       sum = sum + value[i];
    mean = sum / count;
    printf("Mean value is %f\n", mean);

    /* Calculate the standard deviation */
    sum = 0.0;
    for ( i = 0; i < count; i = i + 1 )
       sum = sum + powf(value[i] - mean, 2.0);
    stddev = sqrt(sum) / count;
    printf("Standard deviation is %f\n", stddev);
}
```

This program introduces a number of new and important concepts – dealing with files, return values from functions, conditional execution, error checking, indefinite loops and the use of the C pre-processor to represent constant values.

***Exercise 12**: Modify* `prog4.c` *to create* `prog5.c`*. Compile* `prog5.c`*. Create a data file called* `file.dat` *in emacs containing some numeric values. Run* `prog5` *so it calculates the mean and standard deviation of these values.*

Input/output using files

To use a file we need to perform three basic operations in sequence: to open the file (for reading and/or writing), to read or write our data, and then to close the file when we are finished with it.

Every computer operating system allows files to be given *filenames*, which are used by the user to identify and distinguish the files on disc, however different operating systems use different conventions to construct the filename and represent its location within a directory structure.

The `fopen` function is used to open a file and associate a *filehandle* with it. In the example program above it is used as follows:

```
FILE *fh;
fh = fopen("file.dat", "r");
```

The first argument to `fopen` is the name of the file, in this case enclosed in double quotes as it is a string literal. The second argument is the *access mode*, in other words whether the file is to be opened read-only (`"r"`), for read/write (`"rw"`), or write-only access (`"w"`). Note that if you open an existing file for write or read/write you will overwrite the existing contents. The value returned by the `fopen` function is a filehandle, which can then be passed to other input/output functions to instruct them which file to access. A filehandle is a variable of a special type `FILE*`.

Reading numbers from a file is achieved by the function `fscanf`, which is very similar to the function `scanf` used to read values from the keyboard. The only difference is that it has an additional argument, the filehandle representing the file from which the values should be read, for example:

```
fscanf(fh, "%f", &value[count])
```

The filehandle argument slots in before the format specifier; other than that everything is the same as `scanf`. Another point to notice is that the program actually make use of the value returned by the function `fscanf`. The C Standard dictates that the `fscanf` function should return the number of values actually read from the input file. The format specifier we are using is only asking for a single real value to be read from the file, so our call `fscanf` will return the value 1, up until the point where we reach the end of the file and there are no more values to be read. Once this happens `fscanf` will return zero.

The `feof` function simply checks whether we have reached the end-of-file, and returns "true" if we have, or "false" otherwise.

Once the contents of the file have been read, the file can be closed using `fclose`:

```
fclose(fh);
```

Conditional execution (the `if`-statement) and error checking

In our `prog5.c` we are using the function `fopen` to try to open a file on disc and associate a filehandle with it. In the case where `fopen` is unable to open the named file (because the file does not exist, or the access permissions do not allow the file to be read for example) then `fopen` will return the special value `NULL`. Clearly if we cannot open the file then we cannot sensibly read any data, and we should report an error and exit from the program. It is good programming practice to check for such errors, and can save a lot of time for the user by giving a helpful error message. The program uses the following code to check for such an error:

```
if ( fh == NULL ) {
     printf("Cannot open data file\n");
     exit(EXIT_FAILURE);
}
```

The `if`-statement will only execute the compound statement that follows it if the logical expression is "true". In this case, if `f` is equal to `NULL`, an error message is printed and the program exits. A further clause (`else`) can be added to define what will happen if the logical expression is not "true":

```
if ( fh == NULL ) {
     printf("Cannot open data file\n");
     exit(EXIT_FAILURE);
} else {
     printf("The file was opened successfully\n");
}
```

Indefinite loops

In example Program 2 we introduced the concept of a definite loop, which is a loop for which the *trip count* (the number of times the loop executes) was known before the loop started. In this program we use an indefinite loop, where the trip count is unknown prior to the commencement of the loop.

Indefinite loops are very useful; in this example we are reading numbers from a data file, and we don't necessarily know how many numbers will be in the data file when we start reading it. The `while` loop in Program 5 executes

```
while ( ! feof(fh) && count < MAXVAL)
```

in other words while we have not reached the end-of-file, and while we have not filled up the work array `value`. Using this simple technique the program will run successfully on any data file containing up to `MAXVAL` values.

Symbolic constants and the pre-processor

We have seen how the C language provides *variables* to represent values that can change throughout the program. Sometimes you will find yourself wanting to use constant values in your program. Many constants will have a special meaning, for example π, or the conversion factor from degrees to radians. You could type these numbers in every time you need them, but this is tedious, error-prone and doesn't make for very readable code. A better solution is to use a *symbolic constant*, eg. `PI` or `DEG_TO_RAD`.

The C language (unlike FORTRAN or C++) does not intrinsically provide support for such named constants, however there is a simple way to get the same effect by using the C *pre-processor*.

Before a C program undergoes compilation it passes through a pre-processor phase. You have already seen pre-processor instructions (called *directives*) in all of the example programs above, they begin with a hash sign ('`#`'), for example `#define` or `#include`. The `#define` directive is very useful for defining constants. In the program above we use `#define` like so:

```
#define MAXVALS 100
```

Whenever the C pre-processor sees the character sequence `MAXVALS` it will replace it with the sequence `100`. This means that the line of code which appears:

```
float values[MAXVAL];
```

will be compiled as if it had been written

```
float values[100];
```

Defining constants like this can be very useful. Firstly, as the name suggests, it is not possible to change the value of a constant. If we assign the value of π to a variable, there is a risk that it might inadvertently be changed, which can lead to some very hard-to-find bugs.

In example Program 5 we used a constant to define the maximum size of a working array. Whenever we wanted to loop over the whole array we used the constant name rather than explicitly use the value. This makes for more maintainable code; if we wanted to change the program to handle more data values we only need to change one line, we don't have to go through changing every loop (which is time-consuming and error-prone).

Another important pre-processor directive that you have already encountered is `#include`. The `#include` directive informs the compiler that you wish to use some of the standard library of functions which are available with the C compiler. For example if you want to use input/output functions (such as `printf`, `fprintf`, or `scanf`), you must ensure you have the line

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

close to the start of the file. To use mathematical functions, you should include the `math.h` header file, thus

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

In the example programs you have seen so far we have already included three standard header files (`stdlib.h`, `stdio.h` and `math.h`) that cover all of the requirements of those programs.

The pre-processor has a number of other useful features, for example macros and conditional compilation, but those are beyond the scope of this course.

***Exercise 13**: In `prog5.c` we read up to MAXVALS values from the data file. If there are more numbers in the file, the program will silently ignore them. Modify the program to print a warning message if the data file contains more than MAXVALS values.*

Writing programs

If you have worked through the text above you should have typed in 5 programs which introduce elementary C syntax and structures. You are now ready to write your own programs but please take note of the following advice about programming:

- Always plan what you are going to do before you start typing lines into the computer. A simple flow diagram or notes on the key structures in the program are often very useful.
- Keep notes of what you do. At the very least paste listings of your source code into your notebook and write down any answers the programs give, also in your notebook.
- Use plenty of comment lines so you and others can understand what the program does. Even the simplest programs become impossible to understand when left unattended for a few weeks.
- Keep the layout of the source code neat. Try to develop a uniform style.
- Don't try and get too much code working at once. It is much easier to debug small sections in isolation. Check numerical results by hand using simple input data if necessary.
- If the answers are wrong you can put in temporary `printf()` statements to list out intermediate results and counters to track down where the error is.

- If you get stuck or you are unsure about something always ask for help.

Programming practice

In this final part of the workshop you will be required to write your own program in C. In doing so you will implement many of the concepts you have studied earlier in the workshop.

The data in the table below are taken from the paper "The Velocity-Distance Relation Among Extra-Galactic Nebulae" written by Edwin Hubble and Milton L. Humason in 1931 and published in the Astrophysical Journal. The data show the mean recessional velocity and photographic magnitude of the galaxies in a number of galaxy clusters:

Cluster	Mean velocity	Mean magnitude
Virgo	890	12.5
Pegasus	3810	15.5
Pisces	4630	15.4
Cancer	4820	16.0
Perseus	5230	16.4
Coma	7500	17.0
Ursa Major	11800	18.0
Leo	19600	19.0

The correlation between log of the velocity and the mean magnitude is closely linear. Data like these led Hubble to conclude that a galaxys' recessional velocity was proportional to its distance. This law, known as Hubble's law, demonstrates that galaxies are not only moving away from Earth, but from each other too, providing strong evidence that the Universe is expanding.

***Exercise 14**: Write a program to perform a least-squares analysis (also called linear regression) of the data and find the line of best-fit relating log velocity to magnitude. The method of least squares fitting is explained fully in the laboratory script 'Error estimation in first year laboratory experiments'. You will need lines something like the following within your program:*

```
float  sumx = 0.0, sumy = 0.0, sumxx = 0.0, sumxy = 0.0;

/* Perform linear regression on sqrt(h) vs. time */
for (j = 0; j < nd; j = j + 1) {
             sumy = sumy + logvel[j];
             sumx = sumx + m[j];
             sumxx = sumxx + m[j] * m[j];
             sumxy = sumxy + m[j] * logvel[j];
}
del = nd * sumxx - sumx * sumx;
gradient = (nd * sumxy - sumx * sumy) / del;
intercept = (sumxx * sumy - sumx * sumxy) / del;
```

When writing your program remember that the main body of the program should be written within the construct

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

```
int main()
{
              main body of the program goes here
}
```

In many experiments we expect a linear relationship between 2 quantities. With this in mind you should write your fitting routine to be as general as possible, as it will prove to be useful tool during your course. In particular write your program so that it will read data from a general data file (up to some maximum number of data points).

How do your results compare to those obtained by Hubble and Humason:

```
Log10 velocity = 0.202 * mag + 0.472
```

(not all of the data Hubble and Humason used had been included in the table above, so some small deviation should be expected).

This exercise concludes the First Year Programming Course.

Quick C Reference

Variables, types and declarations

The fundamental data types in C are:

`char`	a single character (usually 1 byte of 8 bits)
`short int`	usually 1 byte
`int`	an integer usually 2 bytes
`long int`	a large integer usually 4 bytes
`float`	single precision floating point usually 4 bytes
`double`	double precision floating point
`long double`	high precision floating point

The integer types including char can be qualified by unsigned or signed to determine whether or not a sign bit is included.

`unsigned int`	usually 2 bytes giving range 0 to 65535
`signed int`	usually 2 bytes giving range -32768 to 32767
`unsigned char`	1 byte giving range 0 to 255
`signed char`	1 byte giving range -128 to 127

Derived types are created using the declaration operators:

```
*               pointer, a prefix operator
&               reference, a prefix operator
[]              array, a postfix operator
()              function, a postfix operator
```

Constants

Integer constants are written as:

`9876`	assumed type `int`
`987654321L`	type `long`
`9876543211`	type `long`
`U9876`	type `unsigned int`
`u9876`	type `unsigned int`

Integers can be expressed in octal or hexadecimal:

`0123`	leading zero indicate octal constant
`0x134A`	leading `0x` (zero x) indicates hexadecimal

Floating point constants are written as:

`4.321`	type `double`
`4.3e-5`	type `double`
`3.1f`	type `float`
`3.1e-1F`	type `float`
`3.1l`	type `long double`
`3.1e-1L`	type `long double`

A character string constant is enclosed in double quotes.

```
"This is a character string constant"
```

Reserved identifiers

There is a set of identifiers reserved for use as keywords in C and C++ and these must not be used otherwise.

```
asm     continue   float     new        signed    try
auto    default    for       operator   sizeof    typedef
break   delete     friend    private    static    union
case    do         goto      protected  struct    unsigned
catch   double     if        public     switch    virtual
char    else       inline    register   template  void
class   enum       int       return     this      volatile
const   extern     long      short      throw     while
```

Input and output

```
scanf("%d",&value); // scan (read) standard input for value
printf("value typed %d \n",value); // write to standard output
fgets(line,sizeof(line),stdin); // get character string from file stream
sscanf(line,"%f",&ans); // scan string for value
nc=sprintf(textline,"an integer %d",ival); // write to a character string
```

The complete set of format specifiers is:

```
%d        integer decimal notation
%o        integer unsigned octal notation
%x        integer unsigned hexadecimal notation
%u        integer unsigned decimal
%c        single character
%s        string of characters
%e        floating point (single or double) exponential notation
%f        floating point (single or double) decimal notation
%g        floating point as %e or %f, whichever is shorter
```

The full list of escape characters is:

```
\n        newline
\t        horizontal tab
\v        vertical tab
\b        backspace
\r        carriage return
\f        form feed
\a        alert or bell
\\        backslash
\?        question mark
\'        single quote
\"        double quote
\0        null
\ooo      octal number
\xhhh     hexadecimal number
```

Operators

C has a very rich set of operators. Here is a list of common operators in order of precedence.

```
[]              subscripting                pointer[expr]
()              function call               expr(expr_list)
++              post/pre increment          lvalue++ or ++lvalue
```

```
~               complement                  ~expr
!               not                         !expr
-               unary minus                 -expr
+               unary plus                  +expr
&               address of                  &lvalue
*               indirection (dereference)   *expr
()              cast                        (type)expr
*               multiply                    expr*expr
/               divide                      expr/expr
%               modulo (remainder)          expr%expr
+               add (plus)                  expr+expr
-               subtract (minus)            expr-expr
<               less than                   expr<expr
<=              less than or equal          expr<=expr
>               greater than                expr>expr
>=              greater than or equal       expr>=expr
==              equal                       expr==expr
!=              not equal                   expr!=expr
&               bitwise AND                 expr&expr
^               bitwise exclusive OR        expr^expr
|               bitwise inclusice OR        expr|expr
<<              left shift bits             expr<<shift
>>              right shift bits            expr>>shift
&&              logical AND                 expr&&expr
||              logical inclusice OR        expr||expr
?:              conditional expression      expr?expr:expr
=               simple assignment           lvalue=expr
,               comma (sequencing)          expr,expr
```

In this table *lvalue* is an entity which can appear on the left hand side of an assignment, typically a variable name. This may be a simple variable, an array element or a pointer. You should be careful that an lvalue is what you intend, a pointer or a primitive type. The type of both sides of an assignment should be the same.

If you look at some C code you will often see composite operators like '+=' which means add and assign. These can be confusing and I suggest to begin with you avoid using these.

Compiler directives

```
#include <sys/pci.h>      // include a system header file
#include "pciadc.h"       // include a local header file
#define DEVICE_ID 0x0adc  // define a replacement macro
```

Commonly used macro definitions in header files:

```
EXIT_SUCCESS   // function integer return OK
EXIT_FAILURE   // funciton integer return not OK
```

Conditional statement blocks

The basic conditional statement block has the form:

```
if(expression1)
          statement1;
else if (expression2)
          statement2;
else
          statement;
```

If the statements require more than one line you must use curly braces to gather together the scope of each conditional:

```
if(expression1)
{
          statement1a;
          statement1b;
          ...
}
else if (expression2)
{
          statement2a;
          statement2b;
          ...
}
else
{
          statementa;
          statementb;
          ...
}
```

In either case the statement or statements following the (expression) are executed if the value of the expression is true or non-zero.

Definite loops

A typical definite loop has the form:

```
        int a[10];
        int i;
        for(i=0;i<10;i++)
        {
                a[i]=i;
        }
```

Indefinite and infinite loops

Indefinite loops come in two forms:

```
while(expression)
{
        body of loop
}

do
{
        body of loop
}
while(expression)
```

In the second variant the body of the loop is executed before the expression is evaluated thus ensuring the body is executed at least once.

An infinite loop can be set up using:

```
         for(;;)
         {
                 ...
                 if(finish loop expression)
                         break;
```

```
            ...
    }
```

Such a loop should be terminated using break as shown or return. The break statement can be used to terminate any for(), while() or do structure.

Functions and header files

The main program function: the integer `argc` is the number of command line arguments which are passed in character string array `argv`. Note `argv[0]` is the name of the program as it occurs on the command line:

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

The prototype declarations of all functions are held in header files. There are an enormous number of these files and an even larger number of prototype function declarations. The system wide header files are usually found at `/usr/include` on Unix and Unix-like systems. For example the floating-point mathematics functions are declared in `math.h`.

To use any of the functions you must declare them by including the appropriate header file at the top of your source file:

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

List of mathematical functions in C

Here is a list of some of the mathematical functions available in C. If you wish to use any of the following mathematical functions in your program, you will need to ensure that you have the line

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

in the first few lines of your program, and that you are using the `-lm` compiler switch when compiling your code.

`sin(x)`	Sine of x (x in radians)
`cos(x)`	Cosine of x (x in radians)
`tan(x)`	Tangent of x (x in radians)
`asin(x)`	Arcsine of x (result lies between $-\pi/2$ and $+\pi/2$)
`acos(x)`	Arccosine of x (result lies between 0 and $+\pi$)
`atan(x)`	Arctangent of x (result lies between $-\pi/2$ and $+\pi/2$)
`atan2(x, y)`	Arctangent of y/x (result lies between $-\pi$ and $+\pi$)
`exp(x)`	Exponential function
`log(x)`	Natural logarithm (base e) of x
`log10(x)`	Logarithm to base 10 of x
`powf(x, y)`	x to the power y (x^y)
`sqrt(x)`	Square-root of x
`fabs(x)`	Absolute value of x
`fmod(x)`	Returns the remainder of x/y, with the sign of x
`ceil(x)`	Returns the smallest integer not less than x

`floor(x)`	Returns the largest integer not greater than x

In the above table, 'x', and 'y' are of type `double`. All the above functions return `double` results.