An Introductory Course on C

Lecturer: Dr M. Brown

Semesters 1 and 2 1996/97

Department of Electronics and Computer Science

University of Southampton

Abstract

These notes were written to help students, who had little or no previous computing experience, learn the ANSI C programming language. The complete ANSI C programming language is *not* covered in these notes, rather the author has concentrated on what he considers to be most useful. They were originally written for engineering students, but rather than concentrating on one problem domain, the author has tried to provide a general set of introductory notes that will be useful for students with different backgrounds. However, it is strongly recommended that you make use of at least one other source of material (e.g. a reference book) which covers the full ANSI C language.

Id like to thank the following people (in no particular order) who have contributed to this document: Neil Matthews, Jon Roberts, Ross Richardson, Jutta Degener.

To contact the author, E-mail Dr Martin Brown at:
`mqb@ecs.soton.ac.uk`
and these notes, together with other course material, can be obtained from:
`http://www-isis.ecs.soton.ac.uk/computing/c/Welcome.html`

Modifications to version 1 of this document include:

- a short summary section at the end of each chapter,
- the ANSI C standard libraries are included in an appendix,
- the keywords of the C language are included in an appendix,
- several chapters (introduction, functions, pointers) etc., have been overhauled,
- more html links have been included (correct at the time of writing) and
- table of contents is now included in the postscript version.

As usual, some errors have been removed and some points clarified.

This document is *Version 2*.

Contents

1 Introduction

1.1 Computers

Electronic computers can be traced back to the 1940's, although some famous mechanical "calculators" were developed by:

- **Pascal** in 1641, whose machine was capable of performing addition and subtraction;
- **Leibniz** in 1674 mechanised multiplication; and
- **Babbage** in 1840 whose analytical engine was recently rebuilt in London.

Even in these primitive devices, some of the components of a modern computer can be found, such as a temporary memory (register) for holding variables being worked on. Electronic computers were originally proposed as abstract devices which could perform or simulate *any* type of mechanical machine. Their abstract form was proposed by Alan Turing (among others) in the mid 1930s, and just like many other technologies the Second World War provided a stimulus for developing machines which could calculate ballistic trajectories (Colossus and ENIAC) or decyphyer encrypted German messages which had been encoded using the *enigma* machine. In 1948, the worlds first electronic stored program was used in the UK to compute the highest factor of an integer (originally programs were hardwired in the computers which were designed for very specific tasks). Computers have evolved over the years as they have been applied to new and more challenging problems. During the early years of development, computer science was considerably under-hyped:

> *I cannot forsee a world demand for more than 3 mainframe computers.* (Chairman of IBM, 1950s)
>
> *There is nothing to do with computers that merits a Ph.D.* (Max Newman, early 1950s)

and most *serious* academic researchers regarded the subject with some distain. However, during the Seventies, with the invention of silicon VLSI (Very Large Scale Integration) manufacturing techniques, computers became fast, cheap and reliable and this has continued over the past 25 years (NASA was one of the main research agencies responsible for pushing for small, light, reliable computers to use on the Apollo moon shot missions) as their performance continues to rise and the price of computers continues to fall. A brief history of computing is given in Table 1.

1.1.1 What are Computers used for?

Computers have changed the lives of most people in the developed world. During the first twenty years of the computer revolution (1945-1965) they were exclusively used by large businesses and governments, but with the LSI and VLSI manufacturing techniques which were developed in the late Sixties, computing became affordable and began to be applied in more diverse applications.

Many of the current uses of computers are "hidden" because they are embedded in everyday devices. Televisions can use computers to automatically adjust the contrast of the picture depending on the ambient light conditions. Computers are built into cars to actively control the suspension, manage the engine, automatically adjust seating conditions and even warn the driver if they "think" a collision is about to occur.

More conventionally, computers are widely used in Universities to simulate a wide variety of real-world processes: from chemical plants to the expected changes in the eco-system.

Year	Achievement
1939-42	Dr John Atanasoff invented the automatic electronic digital computer.
1946	ENIAC (Electronic Numerical Integrator and Computer) was developed.
1951-59	First generation of computers based on vacuum tubes.
1953	IBM entered the commercial computer market.
1959-64	Second generation of computers based on the transistor.
1964-71	Third generation of computers based on Integrated Circuits.
1963	DEC introduced the PDP-8 minicomputer.
1971 - present	Fourth generation of computers based on Large Scale Integration (LSI).
1971	Unix and the C programming language first introduced.
1977	Apple founded by Steve Jobs and Steve Wozniak.
1981	IBM Personal Computer first introducted.
1984	Apple released the Macintosh with a Graphical User Interface and a mouse.
1989	Windows 3 released by Microsoft for the PC.

Table 1: A brief history of the subject of computing.

This is typically what computers are imagined as being: large glorified calculators which repetatively calculate the answer to the same set of numerical equations.

The way people interact with compters is changing, with the advent of the Graphical User Interface (GUI) and the Internet. GUIs allow non-specialists to use computers easily and make it possible to use Multi-Media applications (text, sound, pictures, etc.) in education. The Internet has also changed the way people communicate, using Email to send messages all over the world, and linking the information stored on computer hard disks into a huge, distributed electronic library.

However, the basic "office" jobs of creating documents using word processing applications, preparing presentations using graphical packages and storing information in spreadsheets and databases are probably the most widely used programs on Personal Computers (PC). This software has evolved in the last 15 years, and is currently, easy to use, relatively cheap and extremely well integrated. An example of such a collection of programs is Microsoft Office.

1.2 Elements of a Computer

A computer can generally be understood as being composed of two parts:

Hardware is the physical parts of the machine such as the central processing unit, screen, disk drives, Random Access Memory etc.

Software are sets of instructions which make the hardware perform operations and calculations etc.

One important piece of software is known as the operating system which is a collection of programs which make it easier for the user to run programs, store files, control the CPU etc. These days, most computers have a GUI which effectivley hides the operating system from the user and files are managed and programs are run by clicking a mouse button, or manipulating an image of a file, for instance. This relationship is illustrated in Figure 1, where an "onion-skin" diagram shows how the user is insulated from the physical hardware by the operating system, and from the operating system by the GUI.

1.2.1 Operating Systems

To run a program on a computer, it must be loaded and then executed, and this is the basic job of an *operating system*. Computer operating systems were originally humans. During

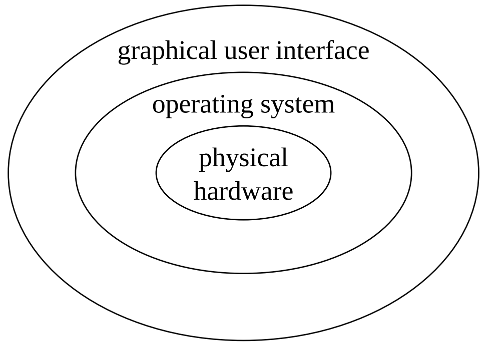

Figure 1: An "onion-skin" model of the relationship between the graphical user interface, the operating system and the physical hardware.

the Fifties, programs were written on punched cards which were placed in a (physical) queue along with other programs and the operator fed each program into the computer while the programmer/researcher went away and had a coffee/lunch. Computers only became interactive with the development of software-based operating systems. Nowadays this is taken for granted and different computing systems operate differing operating systems, although recently there has been a push to develop generic operating systems that can be applied to many different machines (PCs, workstations etc.). This allows different computers to be used in the same way and standardises the tasks that an operating system performs. Once a programmer has written a program it is *compiled* and the operating system is asked to *execute* this set of instructions. Operating systems have now expanded beyond this basic role, and now many commands (small programs) are available for managing your files (editing, searching and allowing access), providing information about the state of the system (which programs are running, who is logged on) and providing access to remote computers via the internet (file transfer, remote logging on etc.).

Operating environments is a term that is sometimes used to describe the combination of an operating system with a GUI. Two of the most common are:

Solaris which is a combination of the Unix operating system and the X windows GUI, and runs on Personal Computers (PCs) and workstations, and

Windows '95 which runs on PCs and used to be based on the PC DOS (Disk Operating System) produced by Microsoft.

You'll be using both operating environments during this course, and to begin with you'll be using the Solaris operating environment running on Sun workstations for C programming. A free variant of this, called Linux, is available for PCs (see introductory course sheet), and this includes the Gnu gcc compiler.

1.2.2 Architectures

A computer is made of many different parts and a high-level schematic diagram is shown in Figure 2. The three main elements are the *storage devices* which can hold large programs and data files, the *computer box* which is composed of the Central Processing Unit (CPU) and its main memory (usually Random Access Memory (RAM)) and the *input/output devices* which allow the user to interact with the computer.

There are many ways of storing large quantities of data, from read-only CD-ROMs (Compact Disk - Read Only Memory) to large Winchester disks (developed at IBM Hursley, near Winchester) which were used by the last generation of mainframe computers. The type of storage used in a system depends on its price, the quantity of data and the required access

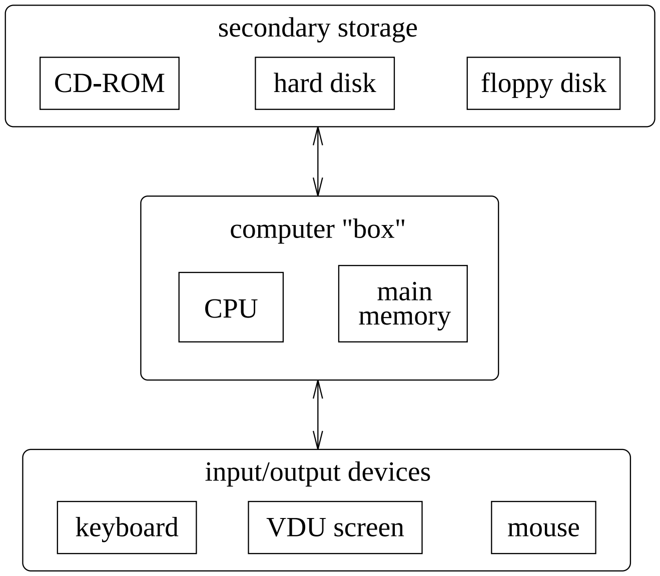

Figure 2: The basic elements which make up a modern digital computer.

time as users always want the cheapest, largest, fastest storage system so the technologies involved in this part of the computer system evolve with time.

For a long time, the input/output devices have been just the keyboard and the Visual Display Unit (VDU) screen, but more recently people have been developing interfaces that are "more natural" such as a point and click mouse (which is standard on most computers now), a pen interface where you can write on special tablets and voice synthesis/recognition systems. Again, computer systems are generally chosen for specific tasks and the keyboard allows you to enter a lot of text data very quickly (at least once you can touch-type), a screen shows you what you are working on and the mouse is a convenient way for choosing between various options in a computer program. Pen-based and voice recognition systems can do without a keyboard as they enable a computer to receive information from words written directly onto the "screen" or by analysing sound. Students often underestimate the importance of storage and input/output technologies, but with GUIs becoming increasingly important, a computer often has a special CPU chip dedicated to displaying information, as well as the main CPU.

The computer's main memory can store all types of data (numeric, symbolic, etc.) and basic instructions. Everything is represented as a set of binary symbols (1s and 0s), and these are stored, retrieved and processed. It is useful to think of each piece of information being stored in a memory cell which can be accessed using its *address*.

The CPU is often what a computer is perceived to be. A PC is often referred to as a "486" or a Pentium whereas this only refers to the type of CPU inside the computer. The CPU is used to process instructions and manage (store/retrieve) the main and secondary storage memory. The CPU can be divided into three main areas: the *control unit*, the *arithmetic-logic unit* and the registers. The control unit, as its name suggests, controls the computer's activities, deciding what instructions to execute and in what order. The arithmetic logic unit performs arithmetic operations (addition, subtraction, multiplication and division) and can also make comparisons, such as determining whether two numbers are equal in value. Instructions are obtained from the *instruction register* and data are read from and stored in the on-chip memory registers. This basic model of a computer's architecture is often referred to as a *von Neumann computer*, after the famous, influential American computer scientist who designed some of the first electronic computers in the Fourties. However, on this course, we shall only be concerned with an even more basic, abstract model of a computer where a

CPU processes instructions and the main memory is composed of a series of blocks that hold the individual pieces of data.

One of the main advances that has been made in computer science is the development of programming languages that are closer to the problem domain than to the hardware on which the programs will be executed. This has enabled engineers, social scientists and mathematicians etc. to design and write programs will little or no knowledge of the physical architecture of a computer.

1.3 Programming Languages

Programming languages are high-level abstractions of the basic operations that can be performed on a *binary*, digital computer. Both the set of instructions that are performed by the computer (the program) and any variables used in its execution are stored in a binary format, and to program in binary is obviously a serious restriction! More readable, 1^{st} generation programming languages were developed in the Fifties and from then on, their capabilities have been extended to make programming easier and more natural. These higher-level languages are then *compiled* to produce an executable, binary file that can be understood by the CPU, as illustrated in Figure 3. Programming languages consist of a restricted set of *keywords* and statements which control the manipulation of the data in the computer's memory.

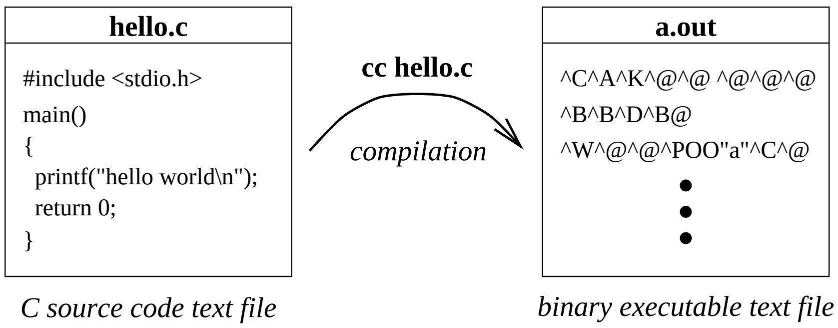

Figure 3: The process of compilation takes an easily understood source file `hello.c` and produces a machine executable binary file `a.out`.

1.3.1 Why C?

Programming languages were first seriously developed in the mid-Fifties and their design was loosely based on the Church-lambda, functional-based, calculus formula proposed in the mid-Thirties. It has been estimated that over 2,000 programming languages currently exist, and these support various programming *styles*. FORTRAN (FORmula TRANslator) was one of the earliest languages and it is still widely used (the most recent version is FORTRAN-90) in industry and academia for large numerical calculations. The latest language specification also supports parallel processing (as well as many ideas directly "borrowed" from C) so its popularity is set to continue. However C, and its newer object oriented derivative C++, have found widespread use throughout industry, and currently they are the most widely used languages. Also, the basic concepts which will be taught in this course can be easily translated to languages such as FORTRAN and Pascal (a popular teaching language created in the Seventies).

The C programming language was inspired by two languages developed in the late 60s called BCPL and its first character-based variant B. It was extensively used in the writing of the Unix operating system (this is what you will be mainly using during this course) and partly because of this, C programs can communicate directly with their environment.

C is unique in being both a *high-level* and a *low-level* programming language, although strictly speaking we could say that C supports both high-level and low-level programming concepts and styles. A low-level programming language is *computer oriented* and could be defined as having the facility to write instructions just as the computer's CPU will carry them out, and they are generally machine specific (i.e. assembler). A high-level programming language is problem-oriented and allows the programmer to structure the program as closely to the problem as possible (i.e. Prolog). This allows less chance for logical design bugs, and although it can be inefficient in terms of running speed, programs can be written quicker and extended (maintained) more easily. C allows programs to be written which use both design concepts and as such is a very flexible language. However, this course will try and stress *good, high-level* C programming style and this will be taken into account when the assignments are marked.

Some of reasons why C is such a popular programming language are because it is *small* and *flexible*. This means that it can be ported to new machines fairly easily (C compilers exist for most popular computers) and also gives the programmer a lot power in how they program. However, it also means that because the language "trusts" the programmer to know what they are doing; it allows you to make mistakes that other languages identify as errors. Sometimes, an experienced programmer may make use of these "features", but for most students, they are a cause of *logical bugs*. During this course we'll highlight some of the most common mistakes and outline programming techniques for overcoming them. Despite this weakness, the flexibility of the C programming language and its ability to support large programming projects mean that it will be widely used for years to come.

The final and probably most important reason why you should learn C is that it leads naturally onto the object-oriented languages C++ and Java. C++ is an object-oriented extension of the C language and is currently becoming the most widely valued, and used, language, both in industry and in research. C++ extends some of the ideas found in C such as structures, memory allocation etc. and places them in an object modelling and design framework. Java is a new object-oriented language, developed by Sun, which allows programs to be downloaded across networks (for instance the WWW/Internet) and run remotely on the local computer. It has a C-like syntax, rather than being a direct extension to the language.

A lot of time and money has been invested in the development and support of C libraries, and any time you spend learning C will not be wasted if you intend to become either a researcher or a software engineer.

1.3.2 Elements of a Programming Language

Computers are simple machines which generally execute one instruction after another. Originally, computers were designed to carry out a specific task and its program (set of instructions) was hardwired into its design. During the Fifties however, the concept of a "software" program was born and electronic computers became truly *universal machines* (a term coined by Alan Turing in 1935 which meant that it could simulate any other machine), limited only by their available storage space and speed.

A computer program is therefore just a list of instructions which the processor is able to understand, although another interpretation could be that its a collection of data and the routines to manipulate this data. These procedures, functions or routines are expressed in primitive ideas which a computer can understand and are then executed sequentially and its results are displayed on some output device. Computer programs are, generally, read like a book: line by line from the top of the file to the bottom. Each line contains some instructions for the computer and the programming language has special *keywords* which allow sets of instructions to be repeatedly executed or a decision to be made about which set of instructions to execute. These features:

Loops: the ability to repeatedly execute the same set of instructions.

Decisions: the ability to decide which set of instructions to execute.

are used to control the program's *flow*, not necessarily performing any calculations in themselves, and is illustrated in Figure 4.

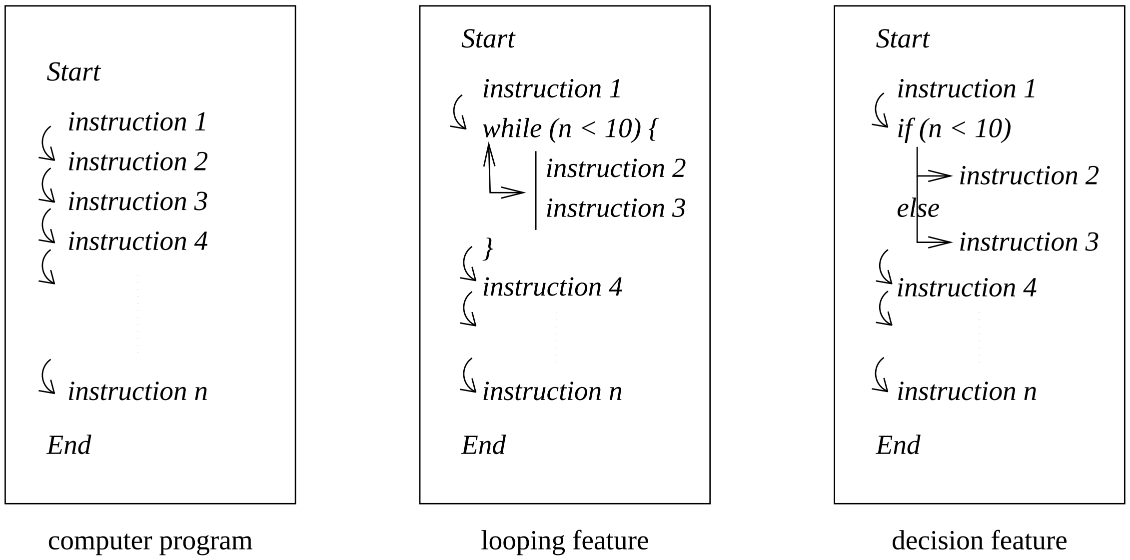

Figure 4: A computer program which is a list of instructions, and the ideas of looping and decision making keywords.

Clearly, the computer program must also be able to instruct the CPU to store, retrieve and manipulate data (where data refers to characters/symbols as well as numbers), and representing this information in the computer's binary memory has the following properties:

Integers exact representation over a limited range, but overflow/underflow may occur when a number is too large or small.

Floating point numbers inexact representation as a finite binary number cannot represent an arbitrary decimal number. However, it can deal (approximately) with much larger numbers than an integer.

Characters and text strings which is a major feature of the C language as it has extensive string manipulation libraries included.

Various operations can be performed on this data:

assignment `=` where the value of the expression on the right hand side is assigned ("remembered by") to the variable on the left.

equality operators `==`, `!=` evaluate whether two pieces of data are the same or not,

relational operators `>`, `<=` etc. compare the the values stored in the data, and

mathematical operations such as `+`, `-`, `*`, `/`, etc. can be performed on the data.

The first two *equality* and *relational* operators are read as questions:

Is the value of the first data equal to the second?

for instance, and this allows the computer to decide when to start and finish loops and also which set of instructions to execute.

1.3.3 Course Synopsis

This course is designed such that the basic concepts of the C programming language will be introduced in the first two weeks using simple examples. These ideas are then discussed in more detail along with investigating how a computer represents a number, how it communicates with the outside world and looking at how programs can be structured to improve their readability and make them easier to maintain. The two (only two?) main problems to learning what a computing language can do:

- Learn how to design programs and how to model real-world tasks.
- Learn what facilities (ideas and keywords) are available in a language.

The work in this course concentrates on the second point, where the:

semantics which are the ideas that a programming language must be capable of representing: iteration loops, logical decisions, recursion etc., and

syntax which is the style or notation a programming language uses to implement its semantics,

will be described and discussed. A limited amount of time will be spent on *algorithmic design* (deciding how to tackle a real-world problem), although this aspect will be partially taught by analysing the structure of the example programs.

1.4 Notation

In this set of lecture notes, all the computer programs and their variables (data) are printed in a typewriter typeface such as `this_is_a_variable`. This allows a C keyword such as `while` to be distinguished from its more normal use. An expression enclosed in angled brackets with a colon such as `<:this is the body of a loop:>` represents a chunk of C code which has been omitted in order to retain generality and to be concise. This will become clear when it is used for the first time, but it should be emphasised that it is *not* part of the C language.

The C programming language uses braces to group together related items (statements, arguments, indexing, etc.) and the following terminology is adopted for the different styles:

`()` *parentheses.*

`[]` *square brackets* or just *brackets.*

{} *curly braces* or just *braces.*

2 Algorithms and the C Programming Language

Computers are built to execute algorithms. The first electronic "computers" built during the Fourties were used to crack codes during the Second World War. Their appearance was totally different to the PCs or workstations we're currently used to, but many of the basic ideas about their architecture are still used. Today, computers still just execute algorithms, but the range of programs (executable algorithms) and hardware (graphical interfaces, virtual reality head displays, sound etc.) that is available, gives the impression that they're totally different from the early machines. In reality, the machines are more powerful (faster and more storage space) and there are an increasing number of ways of interacting with them (mouse, data glove etc.), but their basic functionality and their ability to run algorithms is the same.

2.1 Algorithms

The word algorithm is derived from a famous Arabian mathematican called Muhammad ibn Musa abu Abdallah al-Khorezmi al-Madjusi al-Qutrubulli who was born in about AD 800. His ten books introduced European mathematicians to our present day number system (tens, hundreds etc) and his texts included all the basic arithmetic processes by which numbers could be added, subtracted, doubled, halved, multiplied etc. However, he is probably most famous for the word *algorithm* being named after him.

An algorithm is:

a clear and precise procedure for solving a given problem,

and although the word algorithm evolved from the name of an 8th century Arabian mathematican, the idea pre-dates him by over a millennia. In order to solve a problem on a computer, the corresponding algorithm *must* exist as a series of well-defined steps. A lot of research in computer science is involved with finding algorithms which:

- solve new problems,
- solve old problems more efficiently, or
- produce approximate solutions to problems for which no known algorithm exists.

As has already been mentioned, this course does not concentrate on algorithmic design, but it is considered in sections 11 and 17.

2.1.1 Example: Euclid's Algorithm

In Euclid's 7th book in the *Elements* series (written about 300BC), he gives an algorithm to calculate the highest common factor (largest common divisor) of two numbers x and y where $(x < y)$. This can be stated as:

1. Divide y by x with remainder r.
2. Replace y by x, and x with r.
3. Repeat step 1 until r is zero.

When this algorithm terminates, y is the highest common factor.

As an example, consider trying to find the highest common factor of 34,017 and 16,966. Euclid's algorithm works as follows:

- 34,017/16,966 produces a remainder 85

- 16,966/85 produces a remainder 51
- 85/51 produces a remainder 34
- 51/34 produces a remainder 17
- 34/17 produces a remainder 0

and the highest common factor of 34,017 and 16,966 is 17.

Euclid's algorithm involves several elements:

- simple arithmetic operations (calculating the remainder after division),
- comparison of a number against 0 (test), and
- the ability to repeatedly execute the same set of instructions (loop).

and any computer programming language has these basic elements. The design of an algorithm to solve a given problem is the motivation for a lot of research in the field of computer science, and although this activity will not occupy much time on this course (which will mainly be involved with learning the semantics and syntax of a programming language), we'll return to it a couple of times, half-way through and at the end.

2.2 The C Programming Language

The *semantics* of the C programming language are shared by many popular languages such as FORTRAN and Pascal. Their abilities to store data (numbers and characters) are similar, as are their facilities for repeatedly executing sets of instructions and for making decisions. It is often only the *syntax* of the languages that differ.

Like Euclid's algorithm, a C program is composed of *statements*. A valid C statement is an expression terminated by a semi-colon, and the semi-colon indicates that the CPU should evaluate this statement before moving onto the next one. Generally, each statement occupies a new line, so a C program is read line-by-line, from the top of the file to the bottom. In fact, the body of a C program which implements Euclid's algorithm would look like:

```
int y = 34017;          /* declare and initialise data */
int x = 16966;
int rem;

do {                    /* repeat these actions */
  rem = y%x;               /* calculate remainder */
  y = x;                   /* assign x to y */
  x = rem;                 /* assign rem to x */
} while (rem != 0);     /* as long as rem is non-zero */

printf("The highest common factor of 34,017 and 16,966 is %d", y);
```

Each statement is written on a separate line, each statement is terminated with a semi-colon and the `while` loop terminates (program finishes) when the remainder is zero. The text that occurs between the `/*` and `*/` symbols are notes created by a programmer to remind them what has been written, and as such are ignored by the computer/compiler. The braces `{` and `}` denote the start and end, respectively, of the `do while` loop's body (a compound statement) and these signify which statements should be repeatedly executed as long as the remainder is non-zero. Finally, the `printf()` function writes the answer to the computer's screen once the `do while` loop is finished.

2.3 Summary

- An algorithm is a set of clear and precise instructions for solving a given problem.
- An algorithm must exist before a computer program can be written.
- Writing a program involves knowing both the syntax (keywords) and the semantics (ideas) of the programming language.

3 Your First C Program

3.1 Compilation

Traditionally, the first program every C student learns is one which prints "Hello World" onto the screen. Its useful because it introduces you to a *function* that can write to the screen, `printf()`. To write a program, a *text editor* is used to create (write and save) it and then the program must be *compiled* with the following command:

```
cc hello.c
```

see Figure 3. It can then be *executed* by typing in a command window:

```
a.out
```

The compiler, which is often part of the operating system, reads the C file (which *must* always end with a `.c` extension, otherwise the compiler will not compile the file) and produces another executable file called `a.out` which can be interpreted by the CPU. You can look at the `a.out` file in a text editor if wish, (see Figure 3) and this should make you realise why it is simpler to program using a high level language (such as C) which can then be compiled and executed.

A useful option for the `cc` compiler is to rename the executable file using a more meaningful name, such as `hello`. This is done by compiling the C program with a `-o` option:

```
cc hello.c -o hello
```

and the program can be executed by typing `hello`. This action asks the operating system to submit the set of instructions in the file `hello` to the CPU, which then runs the program. Compilation and execution are two separate actions: one translates a readable set of instructions into their binary equivalent (compilation), and the other instructs the CPU to perform a set of binary instructions.

3.1.1 Which C Compiler?

There are two C compilers available on the Solaris system: the Sun `cc` and the Gnu `gcc` compilers. Despite the fact that the former compiler will be mentioned throughout these notes (because it can be used with Sun's *debugger*), the Gnu `gcc` compiler
`ftp://ftp.ecs.soton.ac.uk/pub/pc/gnu/gcc/`
is freely available for PCs and Unix systems and as such is available to be used at home. In addition, a Unix clone called Linux
`ftp://ftp.ecs.soton.ac.uk/pub/linux/`
is also freely available to be run on PCs. The compiler and operating system are distributed freely by the Free Software Foundation, rather than being tied to a commercial software manufacturer. A wide range of free/shareware programs are available for workstations and PCs and these are generally available over the World Wide Web
`http://www.w3.org/`

To use the `gcc` compiler instead of the `cc` compiler, simply type `gcc` instead of `cc` at the start of the line. The basic compiler options are similar, and the `gcc` compiler could be used to attempt the assignments if you have a PC at home.

3.2 Your First Program

The first program that people have traditionally learnt to one which displays a message onto the screen. This example was first used by Kernighan and Ritchie in the first edition of their book *The C Programming Language*, which first described the C programming language. This involves writing a `main()` function which contains your program and calling a function `printf()` which formats text for displaying on the screen.

3.2.1 Example: `hello.c`

```
/* This is my first program. This part is not looked at by
 * the C compiler because it is commented out.
 *
 * Author: M. Brown
 * Date:   16/9/95
 */

#include <stdio.h>

main()
{
  printf("Hello World\n");
                     /* This function (command) prints the words:
                        "Hello World" to the computer's screen. */
  return 0;
}
```

3.2.2 Formatting the Program

When a C program is typed into a text editor, the program's formatting is not crucial for the compiler, although it is for the programmer! For instance, the above program could be written as:

```
/* This is my first program. This part is not looked at by the C
compiler because it is commented out. Author: M. Brown Date: 16/9/95
*/ #include <stdio.h> main() { printf("Hello World\n"); /* This
function (command) prints the words "Hello World" to the computer's
screen. */ return 0; }
```

but, even for this simple program, it is extremely difficult to determine what the program does or to find bugs (mistakes) when the program does not compile. One example of this is that the C compilers may complain about an error on a certain line number in the file. If this line contains several statements, it is more difficult to identify the error. Generally, each statement should occur on a *separate* line and blocks of statements may be separated by an empty line which the compiler will ignore.

There are many popular styles for formatting C programs, pick one that suits you the best and try and stick to it. A particular set of C Style and Coding Standards produced by SDM is available from (local copy)
`http://www-isis.ecs.soton.ac.uk/computing/c/c_notes/c_style.html`
Points will *not* be awarded in exercises and assignments for original(?) styles.

3.3 Examination of hello.c

3.3.1 Header Files and Libraries

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

The C compiler has a built in pre-processor. Lines that begin with # communicate with the pre-processor. This #include line instructs the pre-processor to include a header file <stdio.h>, which handles the standard input and output. This header file contains the declaration for the function printf() as well as other communication functions. It is important to realise that printf() is not part of the standard C language, but has been included in a library which means that the program can communicate with the outside world. Other functions, such as sin(), sqrt(), can only be used by including the math.h library in a similar manner:

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

(you have to include the -lm compilation flag with the maths library). The angle brackets <> indicate that the header file can be found in the "usual place", which is system dependent. Most of the functions which are contained in the ANSI C standard libraries are listed in appendix B.

C does *not* have any input and output facilities in the basic language and while this may seem like a serious omission, it can be remedied by programmers developing *libraries* of commonly used functions. C has a standard set of libraries which can be accessed by the program designer, two of which are mentioned above. This facility allows programmers to construct their own libraries of commonly used functions, which can then be reused many times in different projects, saving both time and money. Towards the end of this course, you will be writing and using similar libraries.

3.3.2 Comments

```
/* This is a comment */
```

Anything contained between /* and */ is a comment and as such is ignored by the compiler. Comments are notes for the programmer which are *invaluable* for explaining complex pieces of C code. In industry and academia, a lot of time is wasted by programmers unable to understand previously written programs (either their own or someone elses), and in the assignments, marks will be awarded for well commented (*useful* but *concise*) programs. Comments can be placed almost anywhere in a program, but *cannot* be nested, so /*..../*....*/....*/ is invalid; this is an easy mistake to make!

Note it is generally useful to begin the program with one large commented block, which explains the program's operation, describes who wrote it (name and Email address), as well as the date it was last modified. You'll be expected to include this information in the exercises/assignments you submit, so its a good habit to get into at the start of the course.

3.3.3 main() Function

The main() *function* has the general structure:

```
main()
{                                       /* program starts here */
  <:body of main() function: zero or more statements:>

  return 0;                             /* program ends here */
}
```

Every program has a function called `main()` and the parentheses `()` indicate that `main` is a function (this term will be defined later). When a program is executed, it always begins with the first statement in the `main()` function, therefore every C program, no matter how complicated or simple, must have a `main()` function. The braces `{}` that surround the *body* of a function and are used for grouping statements, and every one that lies within these braces is part of the `main()` function. Strictly speaking, every function should return a value, and in this case a `return` value of `0` (zero) means that the program was run correctly. In larger, more complicated programs some functions may return a non-zero value, such as:

```
ans = sqrt(5.5);
```

where `sqrt()` is a function that calculates the square root of 5.5 and the answer is *assigned* to variable `ans`.

Inside the body of the `main()` function, each statement must be terminated with a semi-colon, and forgetting to do so is a common mistake for all types of C programmers (but especially *new* ones).

3.3.4 `printf()` Function

The C system contains a standard library of functions. The include file `<stdio.h>`, see appendix B.9, provides information to the compiler about the function `printf()`, which can then be used many times in the program.

```
"Hello World\n"
```

is a string constant (a piece of text) which is a series of characters surrounded by double quotes. This string is an argument to the function `printf()`, which controls what is to be printed. The two characters `\n` represent a single character called *newline*. It's effect is to advance the cursor to the beginning of the next line. Just like a letter or a number, the request for a newline is a character. This may sound odd, but in a text editor, two lines can be joined together by simply deleting the (invisible) newline character.

The statement:

```
printf("Hello World\n");
```

calls the `printf()` function and passes the string `"Hello World\n"` to it. This is then displayed on the terminal's screen.

3.4 Summary

- The source file for a C program must first be complied before it can be executed. This can be done using the `cc` or `gcc` compilers.

- Every C program must have a `main()` function and the program begins executing at the first statement inside its body.
- Clear and concise comments (text surrounded by `/* ... */`) are invaluable for explaining the action of a program and are ignored by the compiler.
- The `printf()` function allows data to be printed to the screen and it is part of the ANSI C standard libraries.

4 The Distance Conversion Programs

After having compiled and executed a very simple C program, we will now examine some of the language's features that allow more complex tasks to be performed. In particular, we will look at how numbers are stored and manipulated by a digital computer and investigate how looping and decision-making opeators can alter the program's flow. The aim of this section is to expose you to the semantics and syntax of some of the C programming language *without* providing a complete description about how they are used.

4.1 Kilometres - Miles Conversion Program

The aim of this progam is to apply an identical set of instructions (how convert between kilometres and miles) for a range of different integer kilometre values. Therefore, it is necessary to introduce how C handles integer variables, how it performs simple mathematical operations and how a loop can be used to perform the same set of actions a certain number of times.

4.1.1 Example: `distance1.c`

```
/* print kilometres-miles table for kilometres = 0, 10, ..., 100
 *
 * Author: M. Brown
 */

#include <stdio.h>

main()
{
  int kilometres, miles;        /* declare integer variables */
  int lower, upper, step;

  lower = 0;                      /* assign values to variables */
  upper = 100;
  step  = 10;

  kilometres = lower;                          /* initialise kilometres */
                                  /* loop over several statements */
  while (kilometres <= upper) {                /* test kilometres */
    miles = 5*kilometres/8;
    printf("%d\t%d\n", kilometres, miles);
    kilometres = kilometres + step;            /* increment kilometres */
  }

  return 0;
}
```

4.1.2 Program Formatting

Again, the C language does not impose many restrictions on the way the program is formatted, but observing certain rules makes it easier to see the program's structure at a glance. For instance, when a set of commands are contained in the body of either the `main()` function or the `while ()` loop (they're surrounded by a set of braces `{}`) the usual convention is to increase their indentation by two or four spaces. This shows that the statements are related and makes it easier to check that the braces occur in the correct place.

4.1.3 Compilation

To compile the program `distance1.c`, enter the following command in a command window:

```
cc -o distance1 distance1.c
```

and this produces an executable file called `distance1`.

4.2 Examination of `distance1.c`

You should recognise the `main()` and `printf()` functions and the `stdio.h` header file from the previous example, however there are some important new ideas introduced in this program and these will now be discussed.

4.2.1 `int` Declaration

The `int` keyword declares the variables `kilometres`, `miles`, `lower`, `upper`, and `step` to be of type integer, i.e. they can only take integer values and if assigned a real value it will be *truncated* (rounded down to the nearest value). Like every other statement, they are terminated by semi-colons. This declaration reserves a space in the computer's memory for an integer value and every time the variable is referenced in the program, the current value is fetched from memory and manipulated in the appropriate fashion. Whenever you declare a variable, you should give it a meaningful name as this makes programs more readable.

For clarity, these five variables have been declared in two groups, one of which refers to the current distance values, `kilometres` and `miles`, and another which represent the limits of the table: `lower`, `upper` and `step`. However, they could equally have been declared as:

```
int kilometres, miles, lower, upper, step;
```

or:

```
int kilometres;
int miles;
int lower;
int upper;
int step;
```

or even:

```
int kilometres,
    miles,
    lower,
    upper,
    step;
```

Again it is a matter of style which one you prefer, but remember to keep it *concise* and *simple*.

In the C language, the variable declarations should come at the start of `main()` function (just after the opening brace, `{`) and before any other statements. There are a few exceptions to this rule and we shall examine these as they occur later in the course.

4.2.2 Assignments

The next four statements are straightforward *assignments* for the variables `lower`, `upper`, `step` and `kilometres`. Assigning a value to a variable causes that number to be *copied* into the appropriate part of the computer's memory. The first three assignments are numeric and they behave as expected, whereas the fourth assigns the value represented by the variable `lower` to the variable `kilometres`. The computer retrieves the contents of the memory location represented by `lower` and writes the value in the memory reserved for `kilometres`. The `=` symbol is an *assignment* rather than an equals operator, and it does *not* mean that the two variables reference the same piece of memory or anything similar. Instead, it causes the contents of the right hand side to be copied to the variable on the left hand side. Each variable can be thought of a number written in a unique box (its memory location) and the contents of one can be copied into another.

The C language allows declarations and numeric assignments to be performed in just one statement as is shown in the following example:

```
int kilometres, miles;
int lower = 0, upper = 100;
int step = 10;

kilometres = lower;
```

However, just because the language supports this feature doesn't mean you have to use it. Use whatever looks the simplest and the most understandable.

4.2.3 `while` Loop

The ability to perform the same set of instructions repeatedly is one of the most important parts of any programming language and in this program, the `while` loop is used.

```
while (kilometres <= upper) {
  <:body of while loop: one or more statements:>
}
```

This is the first of the three looping constructs available. In this context, while the value of `kilometres` is less than or equal to (`<=`) the value of `upper`, the statements inside the body of the `while` loop (between the braces `{}`) are executed. As soon as the test condition is false, the statement immediately following the closing brace, `}`, is executed.

The most general form of the `while` loop is:

```
while (<:test expression:>) {
  <:body of while loop: one or more C statements:>
}
```

and the CPU evaluates the `test expression` to determine whether it is *true* or *false*. When the test condition is false, it evaluates to a zero numeric value and the body of the loop is never executed. Generally a value of unity is regarded as being true, but any non-zero value is also equivalent to saying that a statement is true.

4.2.4 Arithmetic Expressions

Two of the three statements which form the body of the `while` loop are integer arithmetic expressions. This is what most people think computing is really about: glorified calculating machines, whereas in reality it is only a small part of computer science. However, it is fundamental in engineering applications, and it works largely as you'd expect it to apart from a couple of points which must be made clear. In the first statement, all the variables on the right hand side are integers, so all the calculations are performed using *truncated* integer subtraction, multiplication and division. Therefore, the statement in the program does not produce the same result as:

```
miles = 5/8*kilometres;
```

as `(5/8) = 0.625` which is truncated to zero using integer arithmetic *before* the multiplication is performed. Hence `miles` is always zero. When using integer arithmetic it is generally a good idea to perform all additions, subtractions and multiplications first before doing any divisions. This is a *common* mistake (especially among beginners) so be careful!

Computers can only do one arithmetic operation at a time, so the C statement:

```
miles = 5*kilometres/8;
```

is broken down into the following steps:

multiply `kilometres` by 5 and store this result in a temporary memory location,

divide this result by 8 using integer arithmetic and store this value, and

assign this temporary value to the variable `miles`.

This apparently simple statement is therefore translated into 3 primitive operations which can be executed by the CPU, as illustrated in Figure 5. Most of the time you won't need to examine the arithmetic expressions in this much detail, but sometimes it is necessary to explicitly state *how* the calculations must be performed. However, the rule is always *calculate the value of the expression on the right hand side and then assign this value to the variable on the left.*

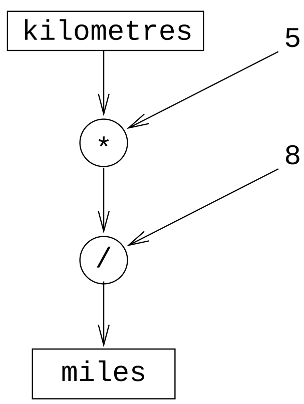

Figure 5: The primitive arithmetic operations generated by the C expression: `miles = 5*kilometres/8`.

The second arithmetic expression is used to increase the value of `kilometres` which is then tested to see if the loop should terminate. It illustrates that the `=` character is an *assignment* operator rather than an equality expression. The CPU first copies the value of `kilometres` out of the memory and places it into its register (a small, fast on-chip memory). The same operation is done for `step` and these two quantities are added together in the register. Finally, this value is used to overwrite whatever was stored at the location of `kilometres` (the `=` operation) and when the statement is terminated, all of the relevant memory in the register is free to be used by the next statement.

4.2.5 `printf()` Again

The `printf()` function used in the `while` loop's body illustrates some more of the features of this versatile function. Here it is used to print out two integers, `%d`, to the screen with a tab spacing, `\t`, in-between. Note that the commands to print out the integers occur in the string `"%d\t%d"` and their values (variable names) occur after the string. If instead, the `printf()` function was written as:

```
printf("kilometres = %d\t miles = %d\n", kilometres, miles);
```

the CPU would treat the first occurrences of `kilometres` and `miles` as *text strings*, and the ones following the string refer to the variables. In this case, the output displayed on the screen would look like:

```
kilometres = 0        miles = 0
kilometres = 10       miles = 6
kilometres = 20       miles = 12
kilometres = 30       miles = 18
kilometres = 40       miles = 25
kilometres = 50       miles = 31
```

Another feature of `printf()` which is very important is the ability to *format* the output, i.e. to print a number which is three characters wide (-99 up to 999, as the negative sign is treated as a number). This is achieved by placing an integer between the `%` and the `d` symbols in the character string. For instance, the statement:

```
printf(kilometres = %3d\t miles = %4d\n", kilometres, miles);
```

produces fields three and four characters wide for the `kilometres` and `miles` variables, respectively. Also, all of the numbers are right justified, so that the output would look similar to:

```
kilometres =   0      miles =    0
kilometres =  10      miles =    6
kilometres =  20      miles =   12
kilometres =  30      miles =   18
kilometres =  40      miles =   25
kilometres =  50      miles =   31
```

We shall now look at a program that does the same operations as the one previously described, but it contains several new features in the C language which make its structure preferable.

4.3 Example: `distance2.c`

```
/* kilometres - miles conversion program, using for loop,
 * #define keyword and double data type.
 *
 * Author: M. Brown
 */

#include <stdio.h>

#define LOWER   0         /* set range of loop */
#define UPPER   100
#define STEP    10

main()
{
  int kilometres;         /* declare integer and floating point variables */
  double miles;
                          /* initialise, test and increment in for loop */
  for (kilometres = LOWER; kilometres <= UPPER; kilometres += STEP) {
    miles = (5.0/8.0)*kilometres;
    printf("kilometres = %3d\t miles = %6.1f\n", kilometres, miles);
  }

  return 0;
}
```

4.4 Examination of `distance2.c`

This is very similar to the previous program, although this may not be immediately obvious, as there are only three main differences.

4.4.1 Defining Constants

In the previous program, `lower`, `upper` and `step` were declared as variables but really they were constant values defined when the program was written. The `#define` line defines a *symbolic name* or a *symbolic constant* that can be used within any expression but cannot be changed. What actually happens is a pre-compilation step replaces every occurrence of the name with its declared value, and because this is a pre-compilation command, it generally occurs before the `main()` function declaration. Putting these constants at the beginning of a program is a good idea because they can be found and changed easily. It is common practice to gave these symbolic constants upper-case names (capitals), as this distinguishes them from normal variables which can be altered.

4.4.2 Floating Point Values

The second major change is the introduction of a new number type called `double`. There are two main ways for representing numbers within a computer: as `int`egers or as `float`ing point (decimal) values. There are two basic types of floating point variables: `float` and `double`, and a `double` is twice as accurate as a `float` (and hence requires twice as much memory). For this program, although `kilometres` is always an integer value, the arithmetic conversion process needs a floating point variable to store the decimal answer (rather than a truncated integer value).

As the arithmetic expression now contains floating point numbers such as `(5.0/8.0)`, all of the arithmetic operations deal with, and return, floating point numbers.

Instead of using a `%d` inside the `printf()` function, a `%f` placeholder signifies that you want to print out a floating point number (rather than an integer). The formatting number `6.1` means that no more than 6 numbers should be printed (including sign and decimal point), with only the first decimal digit being shown.

4.4.3 `for` Loop

The final major difference is the use of a `for` rather than the `while` loop. A `for` loop is mainly used for arithmetic counting, whereas the `while` format is usually used for complex logical tests. It should be recognised that either can be used in every case (i.e. the constructs are redundant) but just like the `#define` statements, the `for` loop puts the initialisation, the test and the increment all within its related set of parentheses `()`. This makes it easier to understand how the loop is constructed, rather than having to search though the statements preceding it and inside its body. However, some loops do not map neatly onto this format and this is mainly when the `while` loop is used.

In this example, the `for` loop is constructed as:

```
for (kilometres = LOWER; kilometres <= UPPER; kilometres += STEP) {
  <:body of for loop: one or more statements:>
}
```

This loop construct has three parts inside its parentheses: *initialisation*, *testing* and *incrementing*, all of which are separated by a *semi-colon*. Note the use of the `+=` operator which increments the value of `kilometres` by the value of `STEP` and store the result in `kilometres`. This is equivalent to `kilometres = kilometres+STEP`, and similar expressions exist for subtraction, multiplication and division.

4.5 Decision Making

As was mentioned in the introduction, the ability to make decisions and to make the program's flow dependent on the value of a variable is a fundamental feature of all programming languages. This facility is provided in C using the `if` and `else` keywords. The `if` keyword is followed by a logical expression which is enclosed in parentheses, and its body of statements enclosed in braces. For example, the following code segment tests to see whether a particular test score is less than 40.0%:

```
double test_mark = 35.0;

if (test_mark < 40.0) {
  printf("You've failed this course, see you for the resits!\n");
}
```

Notice here that the braces are optional as the body of the `if` decision consists of just one statement. However, they are necessary when it consists of two or more statements. The expression

```
test_mark < 40.0
```

should be read as a question and if it is true (evaluates to 1), the statements which make up its body will be executed.

This can be used in the distance conversion program we're developing to print out a message after each conversion. Usually, the body of the `if` decision is more substantial but for this particular program, we just want to print out an appropriate message depending on the distance. This is achieved with the following code segment:

```
if (miles < 5.0)
  printf("we can walk.\n");
else if (miles < 30.0)
  printf("better take the bus.\n");
else
  printf("Ill have to drive!\n");
```

where, depending on whether the `miles` is less than 5.0, greater than or equal to 5.0 and less than 30.0, or greater than or equal to 30.0, a different message (string) is printed. In a general `if - else if - else` structure, only one set of statements will ever get executed and once this is done, the program moves onto the statement following the body of the last decision. The complete program is shown in the following example.

4.5.1 Example: `distance3.c`

```
/* kilometres - miles conversion program, printing messages and
 * using if else decisions.
 *
 * Author: M. Brown
 */

#include <stdio.h>

#define LOWER   0
#define UPPER   100
#define STEP    10

main()
{
  int kilometres;
  double miles;

  for (kilometres = LOWER; kilometres <= UPPER; kilometres += STEP) {
    miles = (5.0/8.0)*kilometres;
    printf("kilometres = %3d\t miles = %6.1f\t", kilometres, miles);

    if (miles < 5.0)            /* print messages depending on miles */
      printf("we can walk.\n");
    else if (miles < 30.0)
      printf("better take the bus.\n");
    else
      printf("Ill have to drive!\n");
  }       /* end of for kilometres loop */
```

```
  return 0;
}
```

4.5.2 Keywords

Keywords such as `while`, `for`, `if`, `else`, etc. are part of the syntax of the C programming language and cannot be used as variable names. A full list of such keywords is given in appendix A, and while you won't come across all of these in this course, it is useful to know they exist!

4.6 Summary

- Variables must be declared at the *start* of the `main()` function in order to represent data inside a program. They should have a type (`int`, `double`, etc) and be given meaningful names which reflect the actual quantities they represent, ie.

  ```
  int runs_scored;
  double metres;
  ```

- Values can be assigned to these variables using the `=` (assignment) operator. This causes the value of the expression on the right hand side to be stored in the variable on the left, ie.

  ```
  runs_scored = 87;
  metres =  5.8*79.2;
  ```

- `while` and `for` loops can be used to repeatedly execute sets of instructions.

- `if -  else if - else` decisions can be used to select which set of mutually exclusive instructions to execute.

5 Data Types

Data are the variables (numbers or characters) which are manipulated by the computer program. Information can be stored in a character string (for instance a person's name or address) or in a numeric form (integer or floating point) which can be used in complex mathematical formulae, and the most commonly used data types are shown in Table 2.

Data Types	Explanation
`char`	character
`int`	integer
`float`	single precision floating point number
`double`	double precision floating point number
`void`	applies to functions only

Table 2: A list of the most common data types represented in the C language.

The `int` and the `double` data types have already been introduced in the distance conversion programs, although it is worthwhile explaining briefly how they are stored in the computer's main memory as this gives an indication of the type of compromises made during the system's design.

5.1 Variable Names

To distinguish between different pieces of data inside a computer program, each variable must be given a unique name. This can then be used store data:

```
miles = <:arithmetic expression:>;
```

as well as to access the numerical value stored in its memory location:

```
<:variable:> = 5*kilometres/8;
```

There are many ways for naming a variable inside a computer program and it is (very) good programming practice to give meaningful names (as has been done in the previous examples) as this makes the program more readable and hence easier to understand and maintain.

Strictly speaking, a variable name must be made up of letters, digits and the underscore character `_`. The first character of the variable name must be a letter and the underscore character is often used instead of a white space when the name is composed of two separate words. Upper case and lower case letters are different, so the variables `x` and `X` are distinct. At least the first 31 characters of a variable's name are significant and if your variable names are any longer you'll probably end up with sore fingers typing them in. The C language keywords, listed in appendix A, such as `while`, `int`, `return` are obviously *reserved* and *cannot* be used as variable names, although `while1` would be acceptable.

In C, it is traditional to use lower case variable names such as:

```
int count;
int age_of_keith = 18;
float x, dx1, dx2;
double pi = 3.14159265;
```

as the uppercase variable names are generally used with the `#define` macro command. Whether you use an underscore (_) or a capital letter to separate different words in a variable's name:

```
int age_of_keith = 18;
int ageOfKeith = 18;
```

is largely a matter of style, but pick one style and stick with it. Mixing styles is as bad as not having one!

5.2 Space and Place

When a variable is declared in the C language, a *fixed* amount of binary memory is reserved for storing the variable's value. The computer's memory is composed of binary cells, so the variable is characterised by the *position*, the *representation* and the *amount* of memory used in the computer.

5.2.1 Bits, Bytes and Words

A computer's memory is composed of a number of *bits* (short for BInary digiTS) which can be either on (=1) or off (=0) and the state of these bits uniquely determines the information stored in the computer's memory at any time. If all numbers could be uniquely expressed in a finite binary form (as computers only have a finite amount of memory), there would be no problems in performing numeric calculations, but because this is not the case, we must understand how numbers are stored.

Firstly, a sequence of eight bits is called a *byte* and the storage capacity of any machine is measured in terms of bytes (or these days in mega $\approx 10^6$ or giga $\approx 10^9$ bytes). When a variable is declared with a statement such as:

```
double miles;
```

a fixed number of bytes is reserved in the computer's memory to store the data, and this information can be retreived and overwritten simply by refering to the variable's name `miles` in the program. Different data types need varying amounts of bytes, and Table 3 shows much memory they require on the Unix system. A *word* is either 2 or 4 bytes long (generally 4 on

Data Type	Memory (bytes)
`char`	1
`int`	4
`float`	4
`double`	8

Table 3: The amount of memory used to store different datatypes on a Unix system. Note that although a `float` and an `int` require the same storage space (32 bits), the interpretation of this binary information is very different in both cases.

the Unix system) and an integer is usually stored in a word. In the memory reserved for a variable, its binary representation is stored and whenever this information is accessed by the CPU (read or overwritten), it does so by examining the *address* of the variable in much the same way as the contents of a book can be looked at by searching for its reference number in a library.

5.2.2 Addresses

The *address* of a variable refers to its location in the computer's memory. To access (retrieve/put) the variable's value, it is necessary to know where it is stored. Figure 6 shows how the value stored in a word of memory can be retrieved once its location (and representation) is known. The C language uses the concept of an addresses (we'll come across it much later in the course), and in this respect it can be imagined as a low-level language feature as it allows the programmer to directly manipulate the computer's memory.

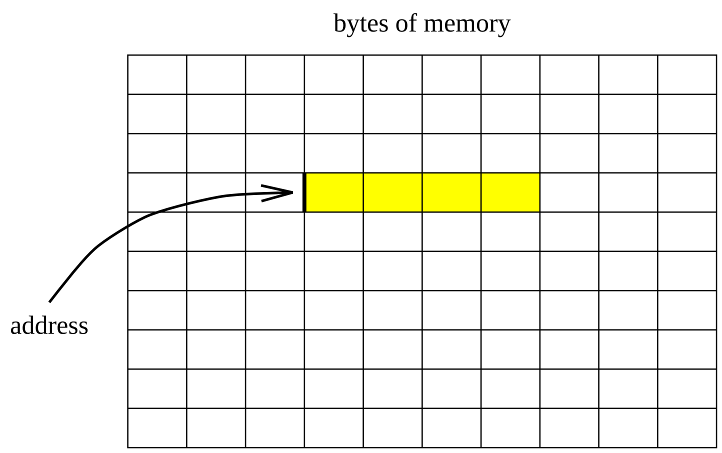

Figure 6: An illustration of how a variable which is stored in a word (4 bytes) is stored and referenced in the computer's memory. Each box represents a byte (8 bits) of memory.

5.3 Binary Representations

So far, we've looked at how much space each data type requires and how it is referenced using its physical address. However, to interpret this binary string correctly, the computer must know what the actual data type is, as the representation of each data type is different.

5.3.1 Integers

Integers can be easily represented in a binary form by simply transforming their base 10 representation to a base 2 form. You probably(?) did this in your basic mathematics course at school. The only problem with representing integers in a computer is that the CPU must request a *fixed* number of bytes for each variable, usually 4, which is the length of a word, and this limits the size of the integer that can be stored correctly. For instance, when an integer is stored using 4 bytes (32 bits) of memory, numbers lying between -2,147,483,648 and 2,147,483,647 ($\approx \pm 2^{31}$, as one bit is reserved for the sign of the number) can be represented exactly and any integer which lies outside this interval would cause an *overflow* or *underflow* if assigned to the variable. Consider the following two statements:

```
int too_large = 123456789123456789;
printf("too_large = %18d\n", too_large);
```

which produce the following output:

```
too_large =        -1395630315
```

Here overflow has occured as the number is too large and positive to store in the variable `too_large`, and the resulting number stored is garbage. Underflow occurs when the number is too large and negative to store in the corresponding variable.

The maximum and minimum sizes of various integers are contained in the header file `limits.h`, see appendix B.4. The minimum value is stored in the symbolic constant `INT_MIN` and the maximum value is stored in `INT_MAX`. The following program prints out these values to the screen and tries to add and subtract 1 to/from their values.

5.3.2 Example: `int_limits.c`

```
/* Print out the minimum and maximum values of an integer from
 * the symbolic constants defined in limits.h.
 */

#include <stdio.h>
#include <limits.h>  /* this library defines INT_MIN and INT_MAX */

main()
{
  int answer;
                 /* print limits of an integer variable */
  printf("Maximum integer value > %d\n", INT_MAX);
  printf("Minimum integer value > %d\n", INT_MIN);

                 /* what happens when 1 is added/subtracted? */
  answer = INT_MAX + 1;
  printf("Maximum integer value + 1 > %d\n", answer);
  answer = INT_MIN - 1;
  printf("Minimum integer value - 1 > %d\n", answer);

  return 0;
}
```

5.3.3 Examination of `int_limits.c`

This program produces:

```
Maximum integer value > 2147483647
Minimum integer value > -2147483648
Maximum integer value + 1 > -2147483648
Minimum integer value - 1 > 2147483647
```

so exceeding the limits by 1 causes the representation to "wrap around" to the other extrema.

5.3.4 Binary - Integer Conversion

To remind you how conventional natural numbers (integers) can be represented as binary strings, consider a binary representation which is only 8 bits (1 byte) long and can store integer values in the range $[-128, 127]$ ($\approx \pm 2^7$). The first (left-most) bit determines the sign of the number (0 is positive, 1 is negative), and the remaining bits represent its magnitude. Table 4 shows the conversion between the two forms.

binary	0	0	1	0	0	1	1	1
power	+	2^6	2^5	2^4	2^3	2^2	2^1	2^0
decimal	$(2^5 + 2^2 + 2^1 + 2^0) = 39$							

Table 4: An 8-bit representation of an integer and its decimal form.

When the leading (sign) bit is 1, the negative representation is calculated by subtracting 2^7 (for this example, but 2^{n-1} in general) from the actual number stored in the remaining bits. Therefore, when the leading bit in Table 4 is 1, its decimal representation is calculated as shown in Table 5.

binary	1	0	1	0	0	1	1	1
power	-	2^6	2^5	2^4	2^3	2^2	2^1	2^0
decimal	$(2^5 + 2^2 + 2^1 + 2^0) - 2^7 = -89$							

Table 5: An 8-bit representation of a negative integer and its decimal form.

On the Unix system, unless you're performing large calculations, integer overflow (underflow) should not occur. The main advantage with declaring a variable as an `int` rather than the more flexible `float` or `double` is that both the representation and all the arithmetic operations are exact, which is *not* the case for floating point numbers.

5.3.5 Floating Point Numbers

Floating point numbers (real/decimal numbers) are represented differently from integers because of the type of information they are required to store. Each real number can be (approximately) expressed in a scientific notation of the form:

$$\begin{gathered} 0.5 \times 10^6 \\ -0.999 \times 10^{27} \\ 0.132 \times 10^{-34} \end{gathered}$$

Each floating point number is characterised by its *mantissa* (the part before the $\times$ sign), its *base* (in this case 10) and its *exponent* (the power to which the base is raised). Not surprisingly, computers use base 2 (binary) arithmetic and by restricting the mantissa to lie in the interval $[0.1, 1.0)$, the exponent is uniquely defined. In fact, because computers use base 2 arithmetic, the scientific notation is really stored as:

$$\text{mantissa} \times 2^{\text{exponent}} \tag{1}$$

Within each slot of memory reserved for a floating point number, the mantissa and the exponent must be stored. Table 6 shows how the memory is used when floating point numbers are stored in 4 and 8-byte memory slots. Generally, the first bit stores the sign of the mantissa, the next set of bits store the exponent (sign and magnitude) and the final bit string stores the magnitude of the mantissa.

Data Type	Length (bits)	Mantissa (bits)	Exponent (bits)
`float`	32	24	8
`double`	64	53	11

Table 6: How the memory is divided between the mantissa and exponent on the Unix system for floating point data types.

When four bytes are used to represent a `float` on Unix systems, the mantissa is accurate to at least 6 decimal places and the exponent (base 10) lies in the range $[-38, 38]$. A `double` data type is similar to the `float` except that the precision of the mantissa is at least 15 decimal places and the range of the exponent (base 10) is $[-308, 308]$. These numbers may give you the impression that the floating point representation is very accurate, however, some numerical calculations are inherently *numerically unstable* and the tiny errors can grow very fast.

On some machines the size of a `double` is equal to that of a `float` and the exact implementation of these terms is machine dependent, but the above discussion is true for the Unix machines you'll be using during this course.

5.3.6 The `long` and `short` of it

As well as having the basic numeric data types `int`, `float` and `double`, the prefixes `short`, `long` and `unsigned` may be applied to an `int` and the data type `long double` is the most accurate floating point number in the basic C language. These qualifiers allow a number to be stored in the most appropriately sized piece of computer memory as the following relationships are always true:

$$
\begin{array}{rcl}
\texttt{sizeof(short int)} \leq & \texttt{sizeof(int)} & \leq \texttt{sizeof(long int)} \\
\texttt{sizeof(signed)} = & \texttt{sizeof(unsigned int)} & = \texttt{sizeof(int)} \\
\texttt{sizeof(float)} \leq & \texttt{sizeof(double)} & \leq \texttt{sizeof(long double)}
\end{array}
$$

where `sizeof` is a C operator which returns the size of the data type in bytes (an unsigned integer).

It is also worthwhile noting that `short`, `long` and `unsigned` are shorthand for `short int`, `long int` and `unsigned int`, respectively, although the data type `double` has no compact form and a `float` is a kind of "`small double`".

You can use these data types to protect the program from incorrect usage. For instance, counting `for` loops generally start at zero and it would be usual to declare the integer index to be of type `unsigned int`. Similarly, if you wrote a piece of code that read in a non-negative number, it would be good programming practice to declare that variable as an `unsigned int` as this would forbid a non-negative value to be entered. However, it is possible that this may significantly increase the complexity of the overall program and implicit assumptions about the range of a variable may be later forgotten about. This will be illustrated in the appropriate laboratory session.

5.3.7 Characters

Just as numbers are represented in a binary form, characters (letters, digits, punctuation marks, white spaces, newlines, etc.) also have a binary representation. Generally, 1 byte is reserved for each character in the computer's character set, so up to 256 (2^8) different characters can be stored. The binary representation scheme that is commonly used is called ASCII (American Standard Code for Information Interchange), and this assigns to each character an eight bit code. Because a single byte can also be though of as an integer in the range $[-128, 127]$ (or equivalently in the range [0,255]), the integers and their character representations are synonymous in the ASCII codebook. Table 7 shows the equivalent integer representation for some of the characters/numbers etc. Each number and letter (upper and lower case) has its own unique ASCII value stored in 1 byte, and a string of n characters is represented internally as a string of n bytes.

A character is initialised by enclosing it in single quotation marks, i.e.:

ASCII Integer [-128,127]	Character
48	0
49	1
⋮	⋮
57	9
⋮	⋮
65	A
66	B
⋮	⋮
90	Z
⋮	⋮
97	a
98	b
⋮	⋮
122	z

Table 7: The ASCII representation of characters on the Unix system.

```
char my_character = 'Z';
```

and the CPU reserves 1 byte of memory for the variable `my_character`. In this is stored the ASCII binary representation which could equally be interpreted as a short integer. Therefore the following `printf()` statement:

```
printf("character = %c\t ASCII = %d\n", my_character, my_character);
```

would produce the following display:

```
character = Z    ASCII = 90
```

This duality between (small) integers and characters can be exploited so that a character could be initialised using the following statement:

```
char my_character = 90;
```

Now the symbol `Z` is stored in the variable `my_character` when it is interpreted as a `char`, and the value `90` is retrieved when it is interpreted as an `int`. Character arithmetic is just like ordinary integer arithmetic, and frequently a character variable is used to represent very short integers when space is at a premium in large programs, although this may be not portable between different computer systems depending on how they represent `chars` ([-128,127] or [0,255]). This can be overcome by explicitly declaring variables as `signed char` or `unsigned char`.

5.4 Summary

- Variables of the data types such as `int`, `float`, `double`, `char`, etc., are stored in the computer's memory as fixed length binary strings. The lengths of the binary strings are determined by the computer system and the data type determines how they are interpreted.

- Integer variables are represented in binary format where the first bit determines the sign of the number and the rest determine its magnitude. Overflow and underflow may occur as the C language does not check the value of the numbers operated on.

- Floating point variables are internally represented in scientific notation (to the power 2), where the mantissa and exponent are stored in binary notation as a single string. As they are stored in a fixed length string, their accuracy is limited and overflow and underflow can occur.

- Character variables can be interpreted as short integers, and they are generally stored using the ASCII representation.

6 Input and Output

Most programs interact with a user, requesting information that changes their future behaviour. This could be as simple as entering a password into the system, or collecting data base entries for statistical analysis. You have already come across the header file `<stdio.h>` and the function `printf()`, both of which define how the program communicates with the outside world (you or external files), and this section describes this aspect in greater detail.

6.1 Formatted Output: `printf()`

`printf()` is a function which displays character strings, integers and floating point numbers on the screen. The examples seen so far have shown how to display results using the `printf()` statement, for simple formatted output. We have already seen the `d` (integer) and `f` (float) format descriptors and Table 8 below shows some of the available format descriptors. After

`printf()`	
Conversion character	How corresponding argument is printed
`%c`	as a character
`%d`	as an integer
`%e`	as a floating point number in scientific notation
`%f`	as a floating point number
`%g`	in `e` or `f` format, whichever is shorter
`%s`	as a string

Table 8: Some of the format descriptors for the `printf()` function.

the percentage sign and before the type descriptor, numbers can be used to control how `printf()` formats the variable. For instance, `%3d` tells `printf()` that the integer should be printed in a field which is at least three characters wide, whereas `%5.1f` is a format descriptor for a floating point number which is a minimum of five characters wide (including its signs and the decimal point) but is accurate to only one decimal place.

6.2 Formatted Input: `scanf()`

The function `scanf()` is analogous to `printf()` in that it reads numbers or characters entered by the user at the keyboard.

6.2.1 Example: `read_int.c`

```
#include <stdio.h>

main()
{
  int x;

  printf("Enter an integer:");
  scanf("%d", &x);
  printf("The integer you entered is: %d\n", x);

  return 0;
}
```

6.2.2 Examination of `read_int.c`

As for the `printf()` statement, the first string is for format control, specifying the type of the information to be read in. The second term, `&x`, is a variable's *address* signified by the `&` symbol. The number typed in response to the `Enter an integer:` prompt, will be stored in the physical memory location used to store the variable `x` ie. at the *address* (`&`) of `x`. The concept of an address was introduced in section 5.2.2 and will be covered in more detail in section 14. For the time being just remember that the variable names in the `scanf()` function need to be preceded by `&` (the compiler won't necessarily complain if you don't do this). Table 9 shows the available format descriptors for `scanf()`. Note that for the `scanf()` function, the format descriptors for `float` and `double` are *different*.

`scanf()`	
Conversion character	What characters in input stream are converted to
`%c`	to a character
`%d`	to an integer
`%f`	to a floating point number (`float`)
`%lf`	to a floating point number (`double`)
`%e`	to a floating point number in scientific notation (`float`)
`%le`	to a floating point number in scientific notation (`double`)
`%s`	to a string, N.B. No `&` symbol on this one

Table 9: Some of the format descriptors for the `scanf()` function.

Note that in functions such as `scanf()` and `fscanf()` (see below), it is important that you use the return value of the `scanf()` function or a debugger to check that the value being read into the variable is correct. `scanf()` returns an integer value which specifies how many pieces of data have been read correctly. If you supply an incorrect formatting statement (such as `%d` when you want to read in a `float`), the compiler won't necessarily check this and the program will run, although the internal values will be incorrect. Sometimes the program will crash with a *segmentation fault*. During program devleopment, you should always check that the values are being read into the variables correctly by displaying their values inside the debugger, possibly using artificial test data.

6.3 File Pointers

Reading information from the keyboard and displaying it to a screen is only one way the program can interact with external information sources. Another method is to read and write information from (and to) files, and although this is achieved with the functions `fprintf()` and `fscanf()` (which are obviously similar to the `printf()` and `scanf()`), the concept of a *file pointer* must first be introduced.

6.3.1 Declaration

Data is read from and written to files using channels opened up between the program and the external environment. To perform this operation is surprisingly simple, but some new concepts must first be introduced. A *file pointer* is a complex data type called `FILE`, but it can be declared like any other data type as:

```
FILE *read;
```

and the only difference is the * which precedes the name of the variable. Basically, it means that the variable `read` is a pointer to a data type `FILE` (this is linked to the concept of an address), and no memory is reserved to store information about the file. This is done when the file pointer is initialised.

A useful analogy for a file pointer is to think of it as an empty (beer) glass which must be filled (initialised) before it can be used properly (drunk), and finally it must be washed (closed) before it can be reused.

6.3.2 Initialisation

A file pointer is initialised, or equivalently a channel is opened between the program and an external file, with a statement such as:

```
read = fopen("file_name.txt", "r");
```

`fopen()` is a function which takes two character string arguments: the name of the file (`file_name.txt`) and an option which specifies whether information is being read from (`r`) or written to (`w`) the file, and returns a file pointer which is used to initialise the variable `read`. If a fault occurs in opening the file (for instance when the file cannot be found), the function returns a value of `NULL`, otherwise `read` can be used to read from and write to this file.

6.3.3 Closing

The function `fopen()` opens a channel between the program and an external file which *must* be closed after all the information has been read from, or written to, the file. This is performed with the following statement:

```
fclose(read);
```

Therefore, it is relatively simple to use files as all that must be done is to declare a file pointer data type, open up the channel and close it when finished.

It is important to remember to close all file pointers, as the information is written to a file using an intermediate *buffer*. Basically, all the information is saved up until the buffer is full, then this block of information is written to the file and the buffer is emptied (this is done for efficiency). Calling `fclose()` ensures that any unwritten output in the buffer is correctly dealt with. This is analogous to the washing up problem at some hypothetical student's digs, where everyone leave the dirty plates until they've run out (buffer is full) and only then wash all the cutlery, plates etc. (empty the buffer). When you have to leave the house (program ends), then it is necessary to wash up everything, even if you've still got some clean cups etc!

6.4 `fprintf()` and `fscanf()`

The functions that read and write numbers/characters/strings from files are similar to `printf()` and `scanf()`, in that all the formatting commands are the same, and the only difference (apart from their name) is that the file pointer must also be passed across as an argument, so that the function knows which channel to use. This is illustrated in the following body of code which opens up two file channels (one for reading, the other for writing) and passes information across these links. Note that the file pointer is always the first term in the parentheses and the character or control string is the second.

6.4.1 Example: file_read_write.c

```
#include <stdio.h>

main()
{
  int x = 5;
  FILE *read, *write;                    /* declare file pointers */

  read = fopen("first_file.txt", "r");    /* initialise file pointers */
  write = fopen("second_file.txt", "w");

  fscanf(read, "%d", &x);           /* read/write using file channels */
  fprintf(write, "x = %d\n", x);

  fclose(read);                         /* close file pointers */
  fclose(write);

  return 0;
}
```

6.5 NULL File Pointers

When the computer is unable to open a specified file, it assigns the `NULL` character to the file pointer. This signifies that an error has occurred and it may be because the user entered the incorrect file name. The following code segment shows how this can occur:

```
FILE *read;

read = fopen("filename.txt", "r");

if (read == NULL)                   /* error occurred, */
  return 1;                         /* end program prematurely. */
```

Here, it was decided to stop executing the program's, if it was unable to open the specified file. Another, more intelligent, course of action would have been to prompt the user for another file name and tried to open that one instead.

Generally, a return value of `0` in the `main()` function means that the program ran correctly whereas any other (integer) value signifies an error. Strictly speaking, you should return the symbolic constants `EXIT_SUCCESS` and `EXIT_FAILURE` which are defined in the library header file `<stdlib.h>`, and note that these conform to the convention that the names of *all* `#define`'d symbolic constants should be in uppercase.

6.6 Summary

- The C programming language does not have any input and output facilities, rather these are supplied by libraries and the basic input and output functions `printf()`, `scanf()`, etc., are declared in the header file `stdio.h`.

- The *addresses* of the variables (`&` in front of the variable name) must be supplied to the `scanf()` and `fscanf()` functions, in order to read data from the keyboard and files, respectively. The formatting options *must* also be correct, and a debugger should be used to check this.

- A `FILE` pointer must be used in order to open a channel between a program and a file, so that data can be read from and written to files. The functions `fopen()` and `fclose()` are used to open and close the channel, respectively.

- If a file channel isn't opened correctly, a `NULL` value is assigned to the `FILE` pointer and this can be used to inform the user that the filename was inappropriate or that no more channels can currently be opened.

7 Operators

Arithmetic operators give the C programming language the ability to act as electronic calculating machines, as at a basic level, they can perform mathematical calculations and store the result in a variable. In addition, relational operators can be used to test whether or not two variables have the same value, and these can be combined into complex logical expressions using logical operators. These operators are used to decide when a loop stops or which set of statements to execute, hence it is necessary to describe them first before the more advanced looping and decision structures.

7.1 Arithmetic Operators

The arithmetic operators such as addition, multiplication etc. often give people the wrong idea that computer science is just about building glorified calculators. However, this was why computers were originally constructed by Pascal (1641) and Babbage (1840), and many researchers run large number crunching programs on todays work stations and supercomputers.

Arithmeitic Operators	Explanation
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus (remainder), integer only

Table 10: The 5 arithmetic operators that are available as part of the C language.

While the effect of most of the operators shown in Table 10 should be obvious (apart perhaps from the modulo `%` operator which calculates the remainder when using integer division), the *order* in which these operators are evaluated needs to be considered as well as the *type* of arithmetic (integer or floating point) being used.

7.1.1 Order of Evaluation

One of the first points to notice when writing complex arithmetic expressions is that white spaces (tabs, spaces, newline characters etc.) have no meaning, as the following two statements are equivalent:

```
y = x*x + 2*x - 5*x/y;
y = x * x+2 * x-5 * x/y;
```

Humans sometimes use white spaces instead of parentheses, `()`, in order to force an expression to be evaluation in a certain sequence, whereas parentheses are absolutely necessary in a computer program if the expression does not correspond to its own inbuilt *precedence rules.* An example of this has already been seen in section 4.2.4.

The C language assigns the same precedence to the `*`, `/` and `%` operators, which is higher than that of the `+` and the `-` operators (which again have the same precedence). When operators of the same precedence occur in the same statement, their order of evaluation is from left to right. Therefore the order of evaluation in the above statements would be:

multiplication:	`x*x`
multiplication:	`2*x`
multiplication:	`5*x`
division:	`(5*x)/y`
addition:	`(x*x) + (2*x)`
subtraction:	`((x*x) + (2*x)) - ((5*x)/y)`

where braces `()` are used to denote previously calculated numbers.

Table 11 shows the order of evaluation for all the different operators. This is mostly common sense as multiplication and division is evaluated before addition and subtraction, and if a different evaluation order is required, it is always possible to force any evaluation order using appropriately placed parentheses. However, using too many parentheses can make an expression unreadable (i.e. difficult to interpret and debug).

Operators	Precedence Order
`() [] -> .`	left to right
`! ~ ++ -- + - *` (unary) `&` (type) `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Table 11: Order of evaluation for a range of different C operators. The operators higher in the table are evaluated first.

7.1.2 Data Type Arithmetic

Your second and third programs (`distance1.c` and `distance2.c`) have already introduced the idea that integer and floating point calculations are handled differently. However, there are some simple rules which determine how the CPU deals with integer and floating point arithmetic. As discussed in the previous section, a complex arithmetic statement is broken down into a series of simple binary operations which involve two variables. When both of these variables are integers, integer arithmetic is applied and when both are floating point, floating point arithmetic is used. If one variable is an integer and the other a floating point variable, the integer is converted to a floating point representation and floating point arithmetic is used. However, it is possible to force or *cast* a variable to a different type, as illustrated in the following code segment:

```
int numerator = 5;
int denominator = 9;
double answer1, answer2;
```

```
answer1 = numerator/denominator;
answer2 = ((double) numerator)/denominator;

printf("Performing integer division, answer1 = %f\n", answer1);
printf("Performing double division, answer2 = %f\n", answer2);
```

The `(double)` casting operator causes the integer variable `numerator` to be represented as a `double` in the CPU's register for this arithmetic operation. C's inherent data type casting then causes `denominator` to be temporarily stored as a `double` as well, and double precision floating point arithmetic is performed, storing the answer in the variable `answer2`. Note that the type of the variable on the left hand side of the assignment has *no* influence on the evaluation of the expression on the right hand side. This is because the expression is always evaluated *before* the assignment operation can take place.

Any numeric data type can be cast as any other, but again, rather than trying to force every expression to be correct explicitly, it is normal to rely on C's built in rules, otherwise the expression can be unreadable.

Using C's integer arithmetic conversion rules, every floating point value is rounded down (truncated) to the nearest integer value. So the value of `answer1` is `0`, but if you wanted to find the *nearest integer*, you'd have had to use the following statement:

```
answer1 = (int) (((double) numerator)/denominator + 0.5);
```

and this would produce the result `answer1 = 1`.

7.2 Assignment Operators

As well as the simple assignment operator `=`, there are many others that can be used which are shorthand for more complex expressions. It is not essential that you use them, but they can make a program more readable (to an experienced programmer!).

Assignment Operators	Explanation
`=`	simple assignment
`++`, `--`	increment and decrement
op`=`	compound assignment - where *op* is any arithmetic operator (e.g. `+`, `-`, `*`, `/`, `%`)

Table 12: The three types of assignment operators.

The majority of assignment operators contained in Table 12 are straightforward, although the use of the automatic increment and decrement operators may seem initially a little confusing. Also note that the `=` is used in an assignment, e.g. `x = 5`, and `==` is used in a relational expression that yields a true or false result such as an `if` decision, e.g. `if (x == y)`.

7.2.1 The Unary Operators ++ and --

The unary `++` operator can be used for pre or post increment. It always increments its operand by one, the difference is *when* the increment takes place.

```
int x = 4;
int y, z;
```

```
y = ++x;    /* x is incremented before its value is assigned to y.
               After execution, y and x will both  have the value 5. */

z = x++;    /* x is incremented after its value is assigned to z.
               After execution z has the value 4, and x the value 5. */
```

The unary -- operator can be used pre or post decrement. It always decrements its operand by one, and again the difference is *when* the decrement takes place.

```
int x = 4;
int y, z;

y = --x;    /* x is decremented before its value is assigned to y. */

z = x--;    /* x is decremented after its value is assigned to z. */
```

In some cases one can use ++ in either prefix or postfix position, with both producing the same result, e.g. the following two statements:

```
i++;
++i;
```

are both equivalent to:

```
i = i+1;
```

As statements in their own right they are convenient mechanisms for incrementing (decrementing) a variable. In other situations care must be taken as to whether to use pre or postfix notation.

7.2.2 The *op*= Operators

The *op*= operators include +=, -=, *=, /=, %=. These operators are a shorthand way of performing an operation on a variable and assigning the result back to the variable.

```
int x = 4;
int y = 3;

x *= 7;       /* shorthand for x = x*7; */

y -= 6;       /* shorthand for y = y-6; */

x /= y;       /* shorthand for x = x/y; */
```

When there exists an expression on the right hand side of this statement, the expression is *always* evaluated first, and the multiplication assignment statement could be re-written as:

```
x = x * (<:expression:>);
```

for instance.

7.3 Logical and Relational Operators

In order to make decisions within computer programs, there must exist some ways for comparing information and also combining primitive logical expressions into more complex formats. An example of this branching or decision process could be in calculating the roots of a quadratic equation: $ax^2 + bx + c = 0$. When the sign of $b^2 - 4ac$ is positive or zero, there exists two real roots, otherwise the roots are imaginary, and different formulae must be used for each case. This section describes some of these logical operators and how they may be applied.

Relational Operators	Explanation
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`<=`	less than or equal to
`>=`	greater than or equal to
Logical Operators	Explanation
`||`	OR
`&&`	AND
`!`	NOT

Table 13: The set of relational and logical operators.

Table 13 shows all of the relational and logical operators, and the arithmetic relational operators are used just as you'd imagine them. For instance, the expression:

```
x == y
```

is true (1) when the values of `x` and `y` are equal, and false (0) otherwise. The five remaining arithmetic relational operators behave in a similar fashion. Care must be taken to distinguish between the assignment `=` operator and the equality test `==`.

These logical operators evaluate whether or not the variable lies in the the set described by the testing expression. For instance, the expression:

```
x <= 40.0
```

describes the set:

$$A = \{x \in \Re : x \leq 40.0\}$$

and this evaluates to 1 when $x \in A$ and 0 otherwise.

7.3.1 Complex Logical Expressions

The termination of a loop or a branching condition in a C program may not just depend on one expression being satisfied, rather it may depend on multiple criteria and the logical operators, `&&` (AND) and `||` (OR), allow simple expressions to be combined into more complex tests. For instance, when a program prompts a user to enter the percentage mark they receive for this module, the answer must lie in the interval $[0, 100]$. A check could be performed on the number entered by the user in the following manner:

```
if (mark >= 0 && mark <= 100)
             /* correct value entered proceed with program */
```

This expression checks if the percentage mark is greater than or equal to zero AND it is less than or equal to 100. If this combined expression is false (evaluates to 0), the user must reenter this number.
Note: This expression could *not* be written as:

```
if (0 <= mark <= 100)
```

which is legal C code but incorrect. Evaluating the expression in a left to right manner: for any mark greater than (or equal to) 0, the first part will be true and will evaluate to 1 which in turn is always less than 100. This expression will then be true for any value of `mark` greater than or equal to 0. These kind of *logical bugs* in programs are very difficult to trace!

Like an arithmetic operator, the logical AND and OR operators can be strung together to combine several primitive logical expressions. For instance:

```
if ((month == 6 || month == 12) && day == 21)
                    /* summer or winter equinox */
```

would represent an expression which was true if either the month was June (6) or December (12) and the date was the 21^{st} and this would signify that the date was the summer/winter Equinox. In a similar manner to arithmetic expressions, parentheses are sometimes necessary to specify the logical structure correctly when C's inbuilt precedence rules, which are shown in Table 11, are voilated. In the expression above, parentheses *are* necessary, otherwise, due to the precedence rules, it would have been wrongly interpreted as:

```
if (month == 6 || (month == 12 && day == 21))
```

Truth tables can be constructed for the logical AND and OR operators which show how two logical (binary) arguments are combined, as shown in Figure 7. These can be extended (in an obvious manner) to two or more occurrences of the `&&` and `||` operators. Finally, the negation (NOT) operator, `!`, is used to flip the truth value, and is sometimes extremely useful for simplifying complex expressions.

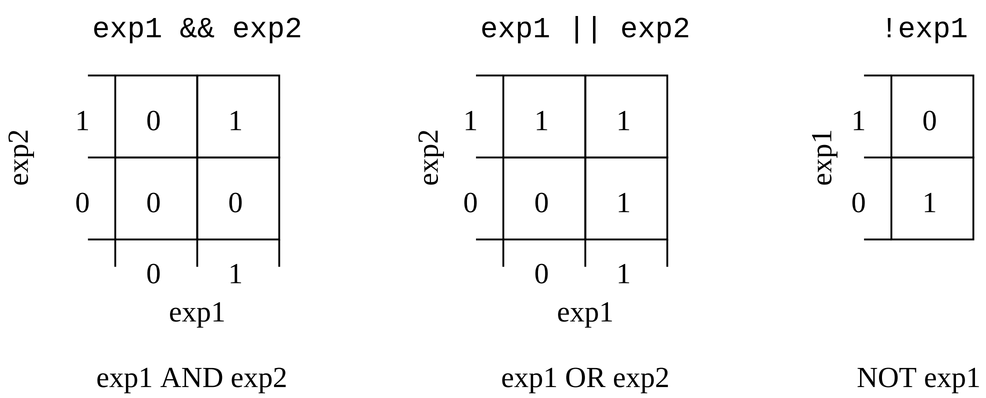

Figure 7: Truth tables for the AND, OR and NOT operators.

Truth tables are used to help evaluate complex logical expressions, as they show the value of applying:

`exp1 && exp2` is true only if both `exp1` and `exp2` are true.

`exp1 || exp2` is true when either `exp1` or `exp2` is true.

`!exp1` is true when `exp1` is false.

7.4 Expressions and Statements

The relationship between expressions and statements is quite simple:

A statement is an expression terminated with a semi-colon!

as the following statements are perfectly legal (assuming the variables have been declared and initialised properly):

```
3.14159265;
a + b;
i <= 10;
```

but don't do any useful work. They simply echo the value of the expression into a temporary working space in memory which is overwritten when the next statement is encountered.

7.5 Summary

- The normal arithmetic operators can be used to evaluate expressions, and their precedence (order of evaluation) is such that `*` and `/` are computed before `+` and `-`.

- The assignment operators are used to change the value of a variable, the simplest being the `=` operators which assigns the value of the expression on the right hand side to the variable on the left.

- The relational operators such as `==`, `<`, `>=`, etc. calculate whether the expression is true (has a non-zero value) or false (value of zero), which can then be combined using the logical operators `||`, `&&` and `!`.

- In C, a statement is simply an expression terminated by a semi-colon. This can cause confusion as beginners often confuse `while (n = 1)` with `while (n == 1)` and both are valid C expressions. However, the former causes the program to get stuck in an infinite loop.

8 Looping structures

The ability to repeatedly process information is one of the most important features in a computer language. For instance, a program may have to calculate the company's payrole for each employee and such a procedure would do similar calculations for each person. Using similar techniques but in a totally different application, many approximate numerical integration routines divide an interval into several subintervals, and an identical calculation is performed for each subinterval. Rather than having to write a set of statements which perform identical operations on different pieces of data, *looping* structures make it possible to repeatedly execute the same set of statements.

There are three looping structures in C, two of which have been seen already, although all three will be shown here for completeness.

8.1 The `while` Loop

The general format of the `while` loop is:

```
while (<:test expression:>) {
  <:body of while loop: one or more statements:>
}
```

and this is illustrated in Figure 8. The expression is *always* evaluated and then the body of the loop is executed only if this is true, else the program moves onto the statement immediately following the loop's body. Once the statements inside the loop's body have been executed, the expression is re-evaluated and the process continues.

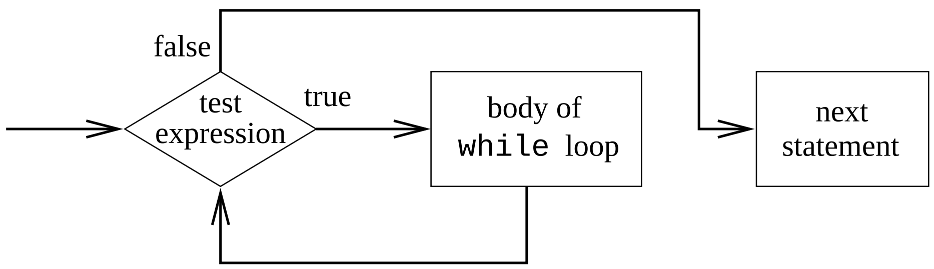

Figure 8: Program flow illustrating the action of the `while` loop.

More formally, it works in the following manner:

- If the expression is true, the statement(s) enclosed in braces `{}` are executed. Note that if there is more than one statement then they must be enclosed in matching braces. There must be at least one statement in the body of the `while` loop, although it could be a *null* statement (a line with only a semi-colon on it).
- When the expression is false, the statement(s) in the body of the loop are not executed, rather the first statement immediately following the closing brace is processed.
- The `while` loop can zero trip, i.e. the statements in the body may never be executed if the `test expression` fails on the first test.

The looping `while` construction is illustrated in the following example.

8.1.1 Example: `while.c`

```
/* Two examples of while loops.
 */

#include <stdio.h>

main()
{
  int index = 0;
                                     /* first while loop */
  while (index <= 10) {
    printf("%d\n", index);
    index++;
  }
  printf("First while loop completed\n");

  index = 0;                         /* second while loop */
  while (index <= 10)
    printf("%d\n", index++);
  printf("Second while loop completed\n");

  return 0;
}
```

8.1.2 Examination of `while.c`

This program performs the following operation twice:

> While `index` is less than or equal to 10, the value of `index` will be printed out and then incremented by one,

and three points to note are:

- The post increment operator, `++`, is used in the second `while` loop as part of another expression. Once the value of `index` has been used in the `printf()` function, it is incremented by one.
- The use of `{}` braces in the first `while` loop to indicate that two statements are to be executed as part of the its body, whereas the second `while` loop's body consists of only one statement so no braces are necessary.
- In both loops, the indentation of the statements lets the programmer tell at a glance which belong in the body of the `while` loop and those which lie outside.

8.1.3 Logical Expression

The `test expression` in the parentheses of the `while` loops in the previous example evaluated the logical expression:

```
index <= 10
```

and depending on whether this was true (1) or false (0), the statements inside the body of the loop were either executed or skipped over. In section 7.3, a number of other relational and equality operators were described as well as the logical operators: `&&` (AND), `||` (OR) and `!` (NOT), which can be used to construct more complex test expressions.

8.1.4 Infinite Loops

Writing an infinite loop in your program means that it will keep running forever, unless stopped by some other means; it does not finish executing with a `return` statement. There are few reasons why you should write an infinite loop, but it is relatively easy to do as shown in the following code segment:

```
while (1) {
  <:body of while loop: one or more statements:>
}
```

The expression `1` is always evaluated as true (it evaluates to its own value), so the statements inside the body of the `while` loop will be executed forever. The only way to stop the program running is to type `^c` (hold the control key down and at the same time type `c`) inside a command window.

It is possible to construct an infinite loop without realising it by typing:

```
while (x = 1) {
  <:body of while loop: one or more statements:>
}
```

Here an *assignment* (`x = 1`) is done instead of a *logical test* (this is a common mistake to make, as has already been noted), and the assignment expression evaluates to the value of the variable. So this expression always evaluates to `1`, which is always true. If your compiler doesn't warn you that there may be an error with this type of expression, see if the computing system has a copy of the program *lint*, and run it over your programs by typing:

```
lint while.c
```

for instance. This should warn you about possible run-time errors rather than compile time errors.

8.2 The `for` Loop

The main problem with the `while` loop is to do with program design. For numerical calculations, a counter must be initialised, tested and incremented (or decremented) and all of these operations occur at different places in the program. Another programmer may find it difficult to work out exactly how the `while` loop is constructed. This is why the `for` loop was introduced which has the following format:

```
for (<:initialisation exp:>; <:test exp:>; <:increment exp:>) {
  <:body of for loop: one or more statements:>
}
```

and this is illustrated in Figure 9.

As an example, the above `while` loop could be written as:

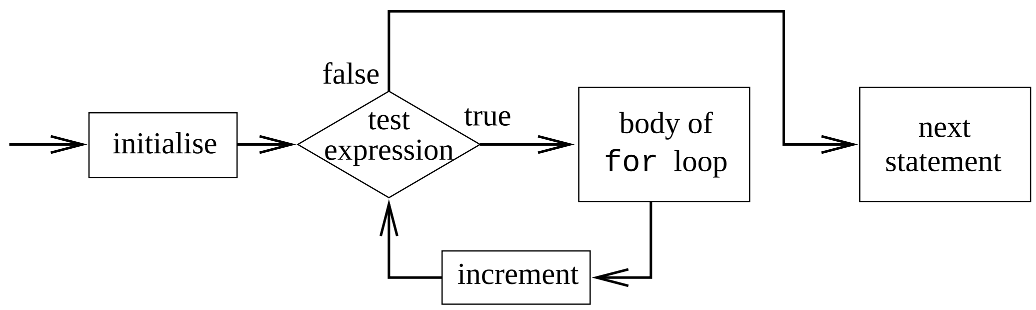

Figure 9: The program's flow within a `for` loop.

```
for (index = 0; index <= 10; index++) {
  <:body of for loop: one or more statements:>
}
```

and as you can see the initialisation, testing and incrementing parts of the loop are all situated together; it is very easy to determine the form of this loop. It should be emphasised that the program's flow is equivalent to the `while` loop, shown in Figure 8. When the program enters the `for` loop, it first evaluates the initialisation expression and then evaluates the test expression. If this is true, the statements inside the `for` loop's body are executed, otherwise the first statement following the `for` loop's body is executed. When the program flow reaches the end of the `for` loop's body, the increment expression is evaluated and the test expression is re-evaluated. This is all illustrated in Figure 9.

The initialisation, testing and step expressions can be empty (but the semi-colons must exist), and two or more initialisation and step expressions can occur (separated by the comma operator) in the `for` loop. This is illustrated in the following code segment:

```
for (i = 0, j = 1; i <= 5 && j > 3; i++, j+=2) {
  <:body of for loop: one or more statements:>
}
```

where two integers `i` and `j` are initialised, tested and incremented within the same `for` loop.

Note you cannot have more than one `test expression`, as this is used to decide when to finish the loop. Also, using using multiple initialisation and incremeting expressions inside the `for` loop's parentheses can sometimes make it difficult to read. It may be "better" to initialise some of the variables just before the `for` loop.

8.2.1 Example: `for.c`

```
#include <stdio.h>

main()
{
  int index, y;
                      /* for loop with a single statement inside its body */
  for (index = 0; index <= 10; index++)
    printf("%d\n", index);
  printf("First for loop completed\n");
                                                  /* more complex for loop! */
```

```
  for (index = 5, y = 1; index > 0 && y < 10; index--, y += 3) {
    printf("index is %d\n", index);
    printf("y is %d\n", y);
  }
  printf("Second for loop completed\n");

  return 0;
}
```

8.2.2 Examination of `for.c`

- The first `for` loop has only one executable statement associated with it and thus does not require the use of braces `{}`, whereas the second `for` loop's body has two executable statements and these are enclosed in braces to reflect this.
- The second `for` loop shows how the comma operator can be used to allow multiple initialisation and incrementing in the same loop. Note that it is quite difficult to interpret it.
- The `&&` operator represents the logical AND. Thus the two statements will continue to be executed while both `index` is greater than 0 AND `y` is less than 10.
- `y += 3` is shorthand for `y = y+3`.

8.3 The `do while` Loop

The `do while` loop is similar to the `while` loop except that the test comes at the end of the loop's main body and so the statements inside its body are *always* executed at least once. Its format is:

```
do {
  <:body of do while loop: one or more statements:>
} while (<:test expression:>);
```

and the program's flow is shown in Figure 10.

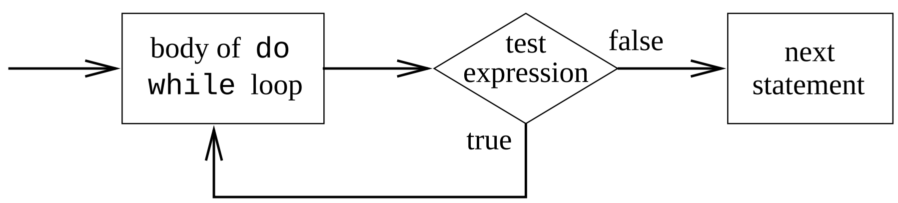

Figure 10: The program's flow around a `do while` loop.

The points made about the previous looping formats also apply here although one common mistake is to forget to put the semi-colon at the end of the `while` parenthese. As illustrated in the following code segment, it is an ideal construct for reading in a value and checking that the entered value is appropriate:

```
main()
{
  int val;
```

```
  do {
    printf("Enter a value between 1 and 10 inclusive: ");
    scanf("%d", &val);
  } while (val < 1 || val > 10);

  printf("Loop completed and value entered is: %d\n", val);

  return 0;
}
```

This loop will continue to be executed while any number less than `1` OR (`||`) greater than `10` is entered. This basic structure is very useful and could be used for checking whether or not a file was opened correctly, as illustrated in the following example.

8.3.1 Example: `check_file.c`

```
/* Use a do while loop to check that the user enters a valid file name
 *
 */

#include <stdio.h>

#define MAX_STRING_LEN   (30+1)

main()
{
  int flag = 0;                        /* never tried to open file before */
  char filename[MAX_STRING_LEN];       /* declare a character string */
  FILE *read = NULL;                   /* file pointer initialised to NULL */
                        /* Now try opening the file ... */
  do {
                        /* Ask for the name of the file */
    if (flag == 0)
      printf("Enter the name of the file (< %d characters):\n",
         MAX_STRING_LEN);
    else
      printf("Error: re-enter the name of the file (< %d characters):\n",
         MAX_STRING_LEN);

    scanf("%s", filename);
    read = fopen(filename, "r");       /* open reading channel to filename */

    flag = 1;                          /* tried at least once to open file */

  } while (read == NULL);

  <:read information and rest of program:>

  fclose(read);
```

```
  return 0;
}
```

8.3.2 Evaluation of `check_file.c`

There are a couple of new keywords and ideas in this program, but the main part of it is contained in the `do while` loop. The integer `flag` stores whether or not the user must reenter the file name and a `0` value signifies that this is the first time that they've had to enter it, whereas a value of `1` means that it has been incorrectly entered. If the file name entered by the user (which is stored in the character string `filename`) does not exist, the value of the file pointer `read` is `NULL` and the user is requested to re-enter the file name.

You may be confused by a couple of points however, and these will be explained now:

- The declaration `char filename[MAX_STRING_LEN];` declares `filename` to be a character array (a *string*) which can store up to `MAX_STRING_LEN-1` characters. For the moment, always make sure that you declare a character array which is sufficiently large to hold the contents of the string read in by `scanf()`. Arrays and strings will be fully discussed in section 10.

- The `if else` decision is used to choose between printing two commands to enter the filename. The integer variable `flag` is used to determine which message to print as it has a value of `0` the first time, and a non-zero value thereafter.

8.4 Summary

- There are three types of loops in the C programming language: `for`, `while` and `do while` loops. Each loop has one or more statements inside its body, and braces *must* be used to group two or more statements together into a compound statement. Usually, the compound statement which forms the loop's body is indented a few spaces for readability.

- `for` loops are particularly suited to mathematical-type operations as the initialisation, testing and incrementing expressions all occur inside its parentheses. A `for` loop can zero trip.

- The `while` loop is similar to the `for` loop except that only the testing expression occurs in the parentheses next to the keyword, and as such it is used in less structured loops. Once again, it can zero trip.

- The body of the `do while` loop is always executed at least once as the testing expression occurs at the end. A semi-colon *must* follow the closing parenthese of the testing expression.

9 Conditional Expressions

Statements in a program are usually executed in sequence, although most programs require an alteration of the flow of control depending on what information the user enters. For instance, consider a program that calculates the roots of the quadratic equation:

$$ax^2 + bx + c = 0$$

There are two distinct situations which can occur: when the roots are real and when they are imaginary. Depending on the sign of the expression $b^2 - 4ac$, you could be taking the square root of a negative (complex solution) or non-negative (real solution) number and different techniques are used to handle these two cases. The `if` keyword can be used to choose between executing various blocks of statements.

9.1 The `if - else` Keywords

The two simplest forms of the `if` statement are: `if` and `if - else`, both of which are demonstrated in the following program.

9.1.1 Example: `if.c`

```
#include <stdio.h>

main()
{
  int val;

  printf("Enter an integer: ");
  scanf("%d", &val);

  if (val > 0 || val < 0)
    printf("The value you entered is non-zero.\n");
  else {
    printf("You entered a zero.\n");
    printf("This is a second print statement\n");
    printf("to show how braces are used!\n");
  }

  if (val > 0 && val < 10)
    printf("Integer entered was a single digit positive number\n");

  return 0;
}
```

9.1.2 Examination of `if.c`

The second `if` is the simplest form of the `if` statement:

```
if (<:test expression:>) {
  <:body of if decision: one or more statements:>
}
```

When the `test expression` is true (evaluates to `1`) then one or more statements in its body are executed, and as before, when there is more than one statement they must be enclosed by braces `{}`. If the expression is false (evaluates to `0`), the statement immediately following its body is executed and this is illustrated in Figure 11. Unlike the looping structures described in section 8, the statements inside the body of an `if` decision only get executed at most once.

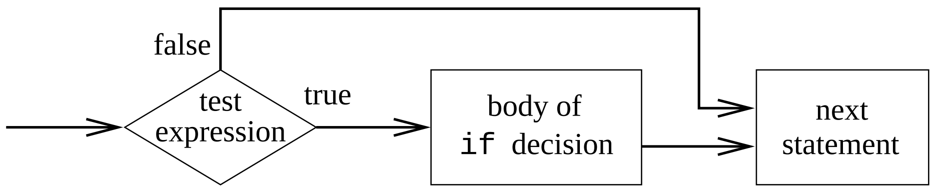

Figure 11: Program flow using the `if` keyword.

In the above example, when the number entered, `val`, is greater than 0 AND (`&&`) less than 10, the `printf()` function will be executed.

The first `if` conditional expression provides an either/or option.

```
if (<:test expression:>) {
  <:body of if decision: one or more statements:>
 }
else {
  <:body of else decision: one or more statements:>
}
```

If `test expression` is true then the statements in its body are executed, otherwise the statements in the body of the `else` decision are executed. In either case, once the appropriate set of statements (body) have been executed, the statement immediately following closing brace of the body of the `else` decision is executed, as shown in Figure 12.

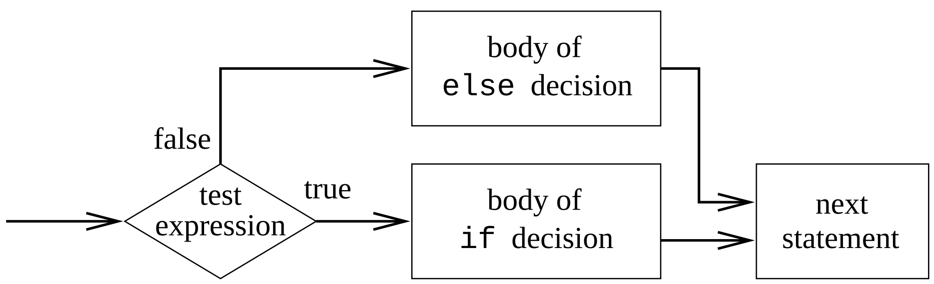

Figure 12: Program flow using the `if else` keywords.

The way that the logical expression is written in `if.c` is correct but *wasteful*, and could be written as:

```
if (val != 0)
  <:non-zero value: single printf statement:>
else {
  <:zero value: three printf() statements:>
}
```

Here, a double relational expression has been replaced by a single one, which is clearer, less prone to mistakes and will probably run faster. The test `val != 0` is equivalent to `!val`, where the second expression uses the fact that a variable echoes its own value and has been used to produce the Boolean value directly. Every non-zero (Boolean) value is regarded as being true (`== 1`) and when `val == 0`, the logical expression `val` is false. These "tricks" can also be used with the looping constructs described in section 8, however they generally make the program less readable and as such won't be used during this course.

9.1.3 `else if` Constructs

Often a program's operation must split between *several* mutually exclusive actions and this can be achieved with several `if - else` constructs, as illustrated in the following example:

```
if (<:test expression1:>) {
  <:body of if: one or more statements:>
}
else if (<:test expression2:>) {
       <:body of first else if: one or more statements:>
     }
     else if (<:test expression3:>) {
            <:body of second else if: one or more statements:>
          }
          else {
            <:body of else: one or more statements:>
          }
```

The set of statements that gets executed depends on which expression is true and when all are false, the statements in the final `else` part of the construct are executed. Obviously, an arbitrary number of these `else if`s can be strung together and generally they are *not* formatted as above but rather like:

```
if (<:test expression1:>) {
  <:body of if: one or more statements:>
}
else if (<:test expression2:>) {
  <:body of first else if: one or more statements:>
}
else if (<:test expression3:>) {
  <:body of second else if: one or more statements:>
}
else {
  <:body of else: one or more statements:>
}
```

which is more readable. This basic format could be used to choose between several options provided to the user (i.e. next screen, previous screen, explain question etc.), although the `switch` conditional expression may be more appropriate, as will now be described.

9.2 The `switch` Conditional Expression

To *switch* between several sets of executable statements in response to the value of an integer variable, the C language has provided the `switch` conditional expression. The `switch` conditional expression is structured like:

```
int option;

switch (option) {
  case 1:
    <:zero or more statements:>
    break;
  case 2:
    <:zero or more statements:>
    break;
  case 3:
    <:zero or more statements:>
    break;
  default:
    <:one or more statements:>
    break;
 }
```

In this example, the value of the integer variable `option` is compared against several values and the appropriate set of statements are executed. The `break;` statement means that the control flow is immediately passed to the first executable statement after the `switch` construct. This can obviously be implemented as an `if - else if - else` construct (where the `default` is equivalent to the final "catch all" `else`) it is just that choosing between several options often occurs in programs (think about how a windows graphical user interface may be implemented every time you press a button) and the `switch` construct along with the `enum` data type can make this programming task easier and more natural.

9.2.1 The `enum` "integer" Data Type

The `enum` data type allows the programmer to use meaningful names instead of obscure integer numbers to represent a set of possible states of a variable. For instance, a set of options on a display may allow the user to go back to the previous screen, go forward to the next one or ask for help. These three actions can be represented as:

```
enum screen_action {prev_screen, next_screen, help};
```

This creates a new data type called `enum screen_action`, and a variable can be declared as follows:

```
enum screen_action user1;
```

Defining a new data type in this manner allows the program to be more readable as the corresponding variables `user1` can be assigned and tested against the declared values, i.e.:

```
user1 = next_screen;

if (user1 == help) {
  <:body of if decision: one or more statements:>
}
```

Therefore, meaningful names can be used instead of the artificial numerical values `0`, `1`, `2` etc., in `if else` and `switch` decisions. Strictly speaking, the C compiler associates an integer with each option (starting from zero and adding one for each option) and the variable itself is another integer. Comparisons are then done by testing for integer equality. However, the program is easier to read using `enum`s, and any technique that can be used to make programming clearer is always encouraged.

9.2.2 The `typedef` keyword

The `typedef` keyword can be used to rename the reserved keywords used in a variable declaration. For instance, the following three commands:

```
typedef int Length;
typedef float Precision;
typedef enum screen_action screen_action;
```

make it possible to use the keywords `Length`, `Precision` and `screen_action` instead of `int`, `float` and `enum screen_action`, respectively, in variable declarations as follows:

```
Length metres, centimetres;
Precision temperature, overdraft;
screen_action user1, user2;
```

The reason for using this command is to make it simpler to declare variables of type `enum`, and generally an `enum` data type is defined and re-named at the same time:

```
typedef enum screen_action {prev_screen, next_screen, help} screen_action;
```

This is the form you should generally use, and it is illustrated in the following example.

9.2.3 Example: `enum.c`

```
/* Program to illustrate  defining an enum datatype and using
 * the corresponding variable.
 */

#include <stdio.h>

typedef enum screen_action {prev_screen, next_screen, help} screen_action;

main()
{
  screen_action user;       /* declare variable of type screen_action */

  user = next_screen;       /* set arbitrary value */

  switch (user) {           /* switch on the variable user */
    case prev_screen:
      printf("Previous screen selected\n");
      break;
    case next_screen:
```

```
      printf("Next screen selected");
      break;
    case help:
      printf("Help option selected");
      break;
    default:
      printf("Error: outside type range for switch\n");
      break;
  }

  return 0;
}
```

Note that the value of the `enum` variable can be read in from the keyboard or a file by using the `%d` formatting string inside `scanf()` and `fscanf()`, and similarly for `printf()` and `fprintf()`. The value which is read and printed is the *integer* representation of the value of the `enum` variable.

9.3 Summary

- The basic way to choose between executing several mutually exclusive sets of operations is using the `if - else if - else` keywords.
- The `switch` construction is often used when the tested variable is an integer.
- Declaring `enum` "integer" data types can make a program *a lot* more readable, and this can be combined with the `switch` conditional expression.
- The `typedef` keyword can be used to rename the `enum` data type definition.

10 Arrays

The basic data types you have come across so far include `char`, `int`, `float` and `double`, as well as the "special integer": `enum`. While these types allow computers to store and manipulate large quantities of text and numeric data, it is inconvenient for the programmer to declare hundreds or thousands of variables of very similar types. For instance, to store my surname I would have to declare and initialise five characters:

```
char surname1 = 'B';
char surname2 = 'r';
char surname3 = 'o';
char surname4 = 'w';
char surname5 = 'n';
```

and to print it out would require 5 `%c` arguments in the `printf()` function's argument list. Obviously, this is not a satisfactory way to represent the data; what happens if I changed my name to another one of a different length, or when this is a list which contained all the surnames of all the lecturers in the university. Similarly, if a program was used to calculate all the student grants in this university, we would need to declare about 8,000 variables like:

```
double student_grant768;
```

Arrays allow you to overcome this problem as you can declare a character array (commonly called a *string*) which consists of five letters as:

```
char surname[6];
```

or 8,000 student grants (of type `double`) as:

```
double student_grant[8000];
```

Arrays allow you to declare and use many related variables of the same data type, and are often used in combination with the `for` loop which allows you to step through each element of the array individually, ie.:

```
  for (i = 0; i < 8000; i++)      /* set all grants to zero */
    student_grant[i] = 0.0;
```

This section investigates how to declare, initialise and manipulate arrays.

10.1 Declaration

An array is an ordered set of data of the same type, where an element can be accessed using the name of the array and its index. In order to declare an array, the following information must be specifed:

data type : the type (`int`, `double`, `char`, etc.) of the elements in the array.

name : which obeys the normal rules for the names of variables.

size : where the number of elements in the array is specified after the array's name inside square brackets.

In addition, an array should be declared at the start of the `main()` function along with the other variable declarations and before any other statement. Therefore, declaring an array is just like declaring a normal variable, except that the size of the array must also be specified.

Examples of array declarations are:

```
main()
{
  int age[10];
  double wages[50];
```

which declares two arrays called `age` and `wages`. The first array consists of 10 `int` variables `age[0]`, ..., `age[9]` and the second array consists of 50 `double` variables `wages[0]`, ..., `wages[49]`.

Often, for readability and ease of change, the sizes of the arrays are `#define`'d at the start of the program. This also allows these symbolic constants to be used to set the limits of the corresponding `for` loops, ie.:

```
#include <stdio.h>

#define MAX_AGE     10
#define MAX_WAGES   50

main()
{
  int i;
  int age[MAX_AGE];
  double wages[MAX_WAGES];
                                        /* set all the ages to zero */
  for (i = 0; i < MAX_AGE; i++)
    age[i] = 0;
                                        /* set all the wages to zero */
  for (i = 0; i < MAX_WAGES; i++)
    wages[i] = 0.0;
```

The symbolic constants, `MAX_AGE` and `MAX_WAGES`, are used in *both* the declaration of the arrays and to set the upper limits in the corresponding `for` loops. If the number of elements in either of the arrays change, it is simply a matter of re-defining the appropriate symbolic constant and all the remaining code should work correctly. This is the form you should generally use for declaring arrays.

10.1.1 Initialisation

Generally, initialising an array must be done on an element by element basis, setting the value of each element individually, as illustrated in the previous code segment where the values of the elements of the two arrays `age` and `wages` are set to zero. However, an array can be initialised and declared at the same time, by using a comma separated list enclosed in braces, ie:

```
#define MAX_AGE     5
#define MAX_WAGES   3
```

```
main()
{
int age[MAX_AGE] = {17, 18, 18, 27, 18};
double wages[MAX_WAGES] = {356.1, 724.9, 741.6};
```

This is useful for small arrays and for character strings (see the next section), but `for` loops should generally be used when it is necessary to set all of the elements to the same value.

Note that when you declare and initialise an array at the same time, it is *not* necessary to specify its size, as the compiler which count the number of elements inside the braces and reserve memory for an array of that size. Therefore, the declarations:

```
int age[] = {17, 18, 18, 27, 18};
double wages[] = {356.1, 724.9, 741.6};
```

are equivalent to the previous set. However, for the reasons mentioned before, it is generally good programming practice to *explicitly* set the size of the array using the `#define` keyword, as this allows the sybolic constant to be used in other `for` loops later in the program, and is a reminder to the programmer about which array is associated with which symoblic constant.

10.1.2 Character Strings

A character string is just an array of characters, except that it is should be `NULL` terminated. This allows the ANSI C functions contained in `string.h` to implicitly know the size of a character array. Character strings can therefore be declared and initialised as:

```
char name[10] = {'A', ' ', S', 't', 'u', 'd', 'e', 'n', 't', '\0'};
```

where the final character `'\0'` is the string terminating `NULL` character. However, this notation is cumbersome, and the C language allows you to declare and initialise a character string in the following manner:

```
char name[10] = "A Student";
```

which is both easier to type and to read. Note that this string is automatically `NULL` terminated, assuming that you have declared enough characters to store the final one (9+1 in this example).

Like numerical arrays, the size of a character array should generally by `#define`'d at the start of the program and it is good practice to explicitly state that you've allowed for the extra `NULL` terminating character in the following manner:

```
#include <stdio.h>

#define MAX_AGE   10
#define MAX_NAME  (128+1)

main()
{
  int age[MAX_AGE];
  char surname[MAX_NAME];
```

The character string `surname` can therefore hold up to 128 characters with room for the terminating `NULL` character.

10.2 Manipulating Arrays

It is relatively simple to retreive and manipulate the values of elements stored inside an array as all that has to be given is the array name and index (inside square brackets). Therefore, the following code prints out the values of the appropriate elements (assuming the arrays have been suitably initialised) of the `surname` and `age` arrays:

```
printf("First letter = %c\n", surname[0]);    /* index is 0 */
printf("Third letter = %c\n", surname[2]);    /* index is 2 */

printf("First age = %d\n", age[0]);
printf("Last age = %d\n", age[MAX_AGE-1]);
```

In the C programming language, the first element of an array of length `n`, has the index `0`, and the last element has the index `n-1`.

Similarly, the following code segment sets the values of the elements directly:

```
surname[0] = 'B';
scanf("%c", &surname[1]);
surname[2] = '\0'

age[0] = 18;
scanf("%d", &age[1]);
```

where it is important to realise that once an index is associated with an array name, the overall variable can be treated just like any other variable. In practice however, arrays are generally used inside `for` loops where the array element's index is an integer variable, ie:

```
#define  MAX_SIZE  10

main()
{
  int index;
  double a[MAX_SIZE], b[MAX_SIZE], c[MAX_SIZE];
                                  /* array a = array b + array c */
  for (index = 0; index < MAX_SIZE; i++)
    a[index] = b[index] + c[index];
```

Here `index` is an integer varible which runs from `0` to `MAX_SIZE-1`, and is used for accessing the individual elements of the arrays `a`, `b` and `c`.

Note it is important to realise in C that the index of the first element of an array is `0`, and the n^{th} element has an index $(n-1)$. Some other programming languages start their array indices at `1`, but this is *not* the case with C. The following program reads in twenty grades in the range 1 to 5, forms the aggregates and prints out the totals for each grade, thus summarising what has been learnt so far.

10.2.1 Example: `array.c`

```
/* Read in MAX_GRADES number of integers in the range 1 -> MAX_NUM
 * and find out how many of each numbers in this range occur.
 */

#include <stdio.h>

#define MAX_NUM       5
#define MAX_GRADES    20

main()
{

  int index, num;
  int count[MAX_NUM];

                         /* initialise the count array to zero */
  for (index = 0; index < MAX_NUM; index++)
    count[index] = 0;

                               /* read in MAX_GRADES numbers */
  for (index = 1; index <= MAX_GRADES; index++) {
             /* check number is in range - if not read again */
    do {
      printf("Enter an integer in the range 1-%d : ", MAX_NUM);
      scanf("%d", &num);
    } while (num < 1 || num > MAX_NUM);

                               /* it's in range so count it */
    count[num-1]++;
  }          /* end of for loop */
                               /* now print out totals */
  printf("Aggregates are:\n");
  for (index = 0; index < MAX_NUM; index++)
    printf("%d's\t%d\n", index+1, count[index]);

  return 0;
}
```

10.2.2 Examination of `array.c`

As illustrated in this example, all array-type operations *must* be done on an individual element basis (apart from declaring and initialising arrays, and reading and printing strings). These operations are generally done within `for` loops as operations are not allowed on the array as a whole, each element must be manipulated individually as in initialising the array to zero. Object oriented languages (for instance C++) allow more complex data types to be created such as an integer array and also allows the operators `+`, `-`, `*`, `/` to be *overloaded* which means that two arrays *can* be added, multiplied etc. In C, if you are expecting to do a lot of array arithmetic, it would be worthwhile creating a library of functions such as `int_add(.)` etc. as this would be equivalent to overloading the standard operators, but not as elegant a solution.

10.2.3 Comparing Arrays

Because an array is not a basic data type, it is not possible to determine whether two arrays have the same value, in the sense that all their elements have the same value, with a simple test such as:

```
#define  MAX_SIZE   10

main()
{
  int a[MAX_SIZE], b[MAX_SIZE];

  <:initialise arrays:>

  if (a == b)
    <:some code here:>
```

In fact, the computer cannot even determine whether these two numerical arrays are the same length as no size information is stored in a numerical array. To test whether two arrays are the same, it is necessary to use a `for` loop:

```
#define  MAX_SIZE   10

main()
{
  int a[MAX_SIZE], b[MAX_SIZE];
  int i, flag = 0;

  <:initailise arrays:>

  for (i = 0; i < MAX_SIZE && flag == 0; i++)
    if (a[i] != b[i])
      flag = 1;
```

When the `for` loop finished, if `flag` is zero, the two arrays are the same in the sense that the values of the corresponding elements are the same.

Note that the compiler will accept `a == b` as a valid C expression, and will not produce an error. This is because it is comparing the *addresses* of the two arrays and determining whether they are the same (in other words, `a` and `b` access the same location in memory). There would be no warning from the `lint` program as well.

10.3 The Character String Library `string.h`

Character stings are simply 1-dimensional arrays of characters with the exception that they are normally `NULL` terminated, and as such require an extra character to be declared. In addition, they can be initialised with a sequence of characters between a pair of double quotes, `"`, at the same time the string is declared. A string can also be read in using the `scanf()` and `fscanf()` functions, but its maximum size *must* be pre-defined and also strings are assumed to be separated by *white spaces*, such as spaces, tabs, line return characters.

It has already been said that standard assignment and logical comparison expressions do *not* apply to arrays in C, and because of this a set of routines have been written, a library, which can compare two strings to see if they're equal, `strcmp()`, to find the length of a string, `strlen()`, to concatenate (join together) two strings, `strcat()`, amongst other useful operations. These functions are declared in the header file:

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

which must be included at the start of your program if you want to use any of these functions. Three of the functions which you can use are:

```
int strcmp(const char *s1, const char *s2);    /* compares s1 == s2? */
size_t strlen(const char *s);          /* returns the length of string s */
char *strcat(char *s1, const char *s2);  /* copies s2 onto the end of s1 */
```

Note that the `const` keyword simply signifies that the argument won't be modified by the function and that `size_t` is an `unsigned` integer data type. The other functions which can be used are described in appendix B.11.

10.4 Multi-Dimensional Arrays

The arrays which have been used so far are 1-dimensional or *vectors*. In many applications however, 2 or higher dimensional arrays are necessary. For example, a 2-dimensional array which stores the surnames (at most 20 characters) of 100 people would be declared as follows:

```
#define MAX_PEOPLE 100
#define MAX_SURNAME_LENGTH (20+1)

main()
{
  char surnames[MAX_PEOPLE][MAX_SURNAME_LENGTH];
```

Three or higher dimensional arrays would be declared in a similar manner by adding an extra set(s) of square brackets on the end, but in most situations you are unlikely to need anything more than a 2-dimensional array.

A declaration such as:

```
int a[3][5];
```

produces a two dimensional array with elements:

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

This array is actually stored in a contiguous piece of the computer's memory as a row of integers with indices:

0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

and it should be clear how 3 and higher dimensional arrays will be stored as a (very) long row of data.

As with 1-dimensional arrays, the standard method of accessing data in a multi-dimensional array is to use (nested) `for` loops, and this is illustrated in the following program.

10.4.1 Example: array2D.c

```
/* Declare and initialise a symmetric 2D matrix and print out the values.
 *
 */

#include <stdio.h>

#define ROW 10
#define COL 3

main()
{
  int row, col;
  int arr[ROW][COL];

                                /* fill array with arbitrary values */
  for (row = 0; row < ROW; row++) {
    for (col = 0; col < COL; col++) {
      arr[row][col] = row+col;
      printf("Row = %d, Column = %d\n", row, col);
    }     /* end for -> col */
  }       /* end for -> row */

                                       /* print out array */
  for (row = 0; row < ROW; row++)
    for (col = 0; col < COL; col++)
      printf("Array[%d][%d] = %d\n", row, col, arr[row][col]);

  return 0;
}
```

10.5 Summary

- An array is declared like a normal variable, by specifying its data type and giving it a name. However, the size of the array (in square brackets) must also be specified, eg:

  ```
  double marks[100];
  ```

  declares an array called `marks` which has `100` elements of type `double`. Usually, the size of the array is `#define`'d at the start of the file and the corresponding symbolic constant is used throughout the remaining program (declaration, `for` loops, etc.).

- The array's elements start with the index `0` and finish with the n^{th}-1 element. The first element in the previous example is `marks[0]` and the last is `marks[99]`. These elements can be treated *just* like normal variables (reading, printing, multiplying, adding, etc.).

- `for` loops are usually used to step through the elements in an array. For instance, the following code sets the value of all the elements in the previous array to `1.0`:

  ```
  for (i = 0; i < 100; i++)
  ```

```
    marks[i] = 1.0;
```

Comparisons between arrays *must* be done on an element by element basis.

- A character string is a 1 dimensional array of characters which is `NULL` terminated. Therefore, an extra character must be declared to hold this value. The ANSI C library functions declared in `string.h` can be used to manipulate character strings.

- 2 (and higher) dimensional arrays can be declared and used with the double (and more) bracket notation. In addition, the elements are accessed using nested loops. For example:

```
double marks[10][20];

for (i = 0; i < 10; i++)
  for (j = 0; j < 20; j++)
    marks[i][j] = 1.0;
```

11 An Introduction to Program Design

In order to design and write computer programs (in any language), it is necessary to have a good understanding of both:

- the elements of a programming language, and
- the definition and structure of the problem/task you wish to solve.

Without both of these aspects, it is impossible to write a computer program which solves a problem or successfully performs some task. So far, you've learnt how to use the *basic* keywords in the C programming language, so it is worthwhile investigating how to analyse a problem and produce a solution which can be translated to a programming language.

11.1 Algorithm Design

Algorithm design, was an topic studied by G. Polya in the late 1940s, and he proposed 4 steps that must be carried out:

1. Understand the problem.
2. Get an idea of how an algorithmic procedure might solve the problem.
3. Formulate the algorithm and represent it as a program.
4. Evaluate the program for accuracy and for its potential as a tool for solving other problems.

For most problems, the designer must adopt an iterative problem solving approach which aims to develop as generic a solution as possible, while keeping it simple and reliable. The designer would gain an initial partial understanding of the problem and then see which algorithms may be appropriate for understanding it. As the designer tries to construct an algorithmic procedure, they would gain a deeper understanding of the original problem so the flow would not always be $1, 2, 3, 4$, rather it could be $1, 2, 1, 2, 3, 2, 3, 4$ as the solutions to different parts of the problem become clearer.

Often parts 1 and 2 are very complex, and for some mathematical problems, their solutions are equivalent to discovering a new theorem before a program can be written. For instance, it is not currently known whether the following procedure (expressed in C) always terminates for an arbitrary initial value:

```
int num;
int step = 0;

scanf("%d", &num);        /* read in user-supplied number */

while (num != 1) {        /* while num is not equal to 1 */
  if (num%2 == 0)         /* if num is even */
    num = num/2;                    /* divide by 2 */
  else                    /* else num is odd */
    num = 3*num+1;                  /* multiply by 3 and add 1 */
  step++;
}

printf("Procedure terminated in %d steps\n", step);
```

and so it is impossible to write a computer program to solve this problem. The description of Euclid's simple algorithm in section 2.1 which calculates the highest common factor of two integers may have given the impression that most, if not all, problems could be solved using computer programs. However, this is far from being true and even when algorithms do exist, it is not always possible to implement them on a computer due to memory and processing time constraints.

11.2 Program Structure

Few general guidelines can be given about how to structure a program, as this depends on the original problem and the algorithm used to solve it. However, some guidelines will be given.

It is often useful to separate a program into three distinct phases:

Initialisation refers to the declaration and initialising of the program's variables at the start of the `main()` function's body (after the opening brace, `{`). It is useful to read all available information *before* processing it, as checks can be made on its accuracy. Running a complex simulation could take hours and it could crash half way through because the user enters an incorrect value.

Data manipulation refers to the major part of the program which contains executable statements, `for` loops, `if else` conditional expressions etc.

Storing results which occurs at the end of the program, and results are printed to the screen/written to a file.

These guidelines may seem obvious to some students, but when you begin to write large computer programs and have to remember what each set of statements do, it is useful to group them together like this and provide brief but descriptive comments in the source file.

11.3 Flow Charts

You've already come across basic flow charts in sections 8 and 9, where they were used to illustrate the operations of the `while`, `for` and `do while` loops and the `if else` conditional expressions (see Figures 8-12). A flow chart is simply a graphical method for representing the program's flow, and is language *independent*; that is it could be used for designing C, Pascal or Fortran etc. programs. Each "box" in a flow chart represents a particular type of instruction, action or data store, and arrows are drawn between the boxes to show how the program flow occurs between statements. Sometimes the arrows are labelled with text to show whether a decision was true or false, see Figure 12, but this should be kept to a minimum. Some of the different box styles are illustrated in Figure 13.

Flow charts encourage the program designer to think about the *structure* and *type* of tests, statements, etc. rather than the language-specific keywords which are used to implement them, and as such they are useful for explaining a program's behaviour to non-experts, and a simple example program is shown in Figure 14.

11.4 Summary

- Program design involves understanding what the problem is and specifying a clear and precise algorithm which can then be written in the most appropriate programming language.

- Program design is generally an iterative process, where parts of the algorithm may be described and the relevant parts of the program implemented before a complete understanding of the problem is achieved.

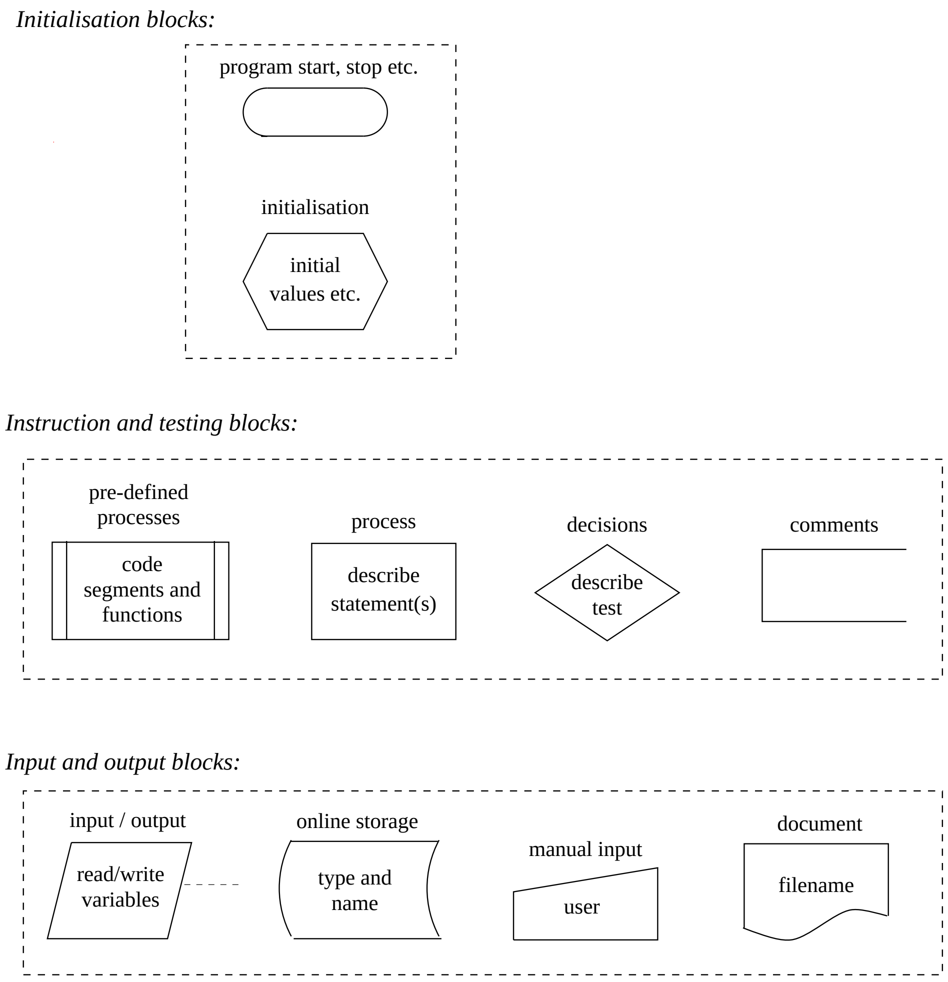

Figure 13: Some of the graphical "boxes" used in a flow chart.

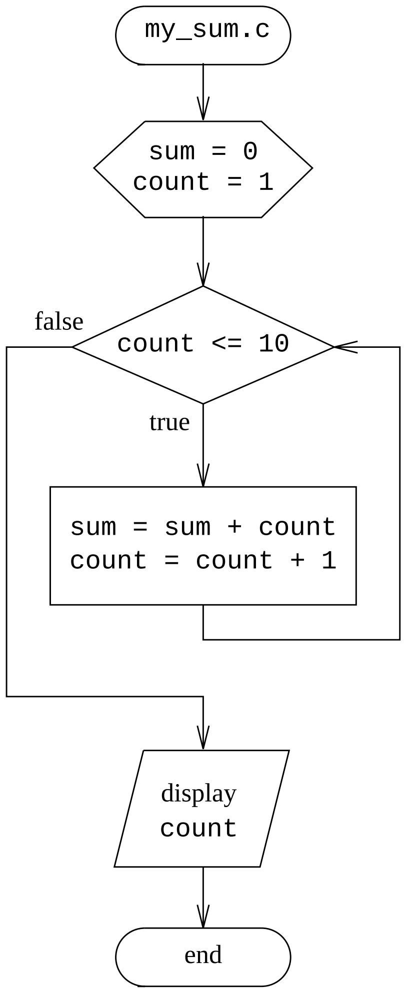

Figure 14: The flow chart for a program which calculates the sum of the numbers from 1 to 10 (inclusive).

- Program structure is largely problem dependant, but it is a good idea to declare and initialise all the variables at the start of the program and printing or storing the results generally occurs near the end. Put comments into the source file to distinguish these blocks.

- Flow charts can be used to graphically describe the basic operations of simple programs.

12 Functions

12.1 Introduction

Functions are used to perform a small job in part of a large project, and you have already met and used several functions:

```
main()            /* every program should have one! */
printf()          /* print data: declared in stdio.h */
scanf()           /* read data: declared in stdio.h */
sqrt()            /* calculate square root: declared in math.h */
fabs()            /* calculate absolute value: declared in math.h */
```

In these examples, the `main()` function is always written by the programmer, but all the others have been designed by another person and form part of the ANSI C standard library, see appendix B. You have used them to perform a particular task, such as printing to the screen or calculating the square root of a number *without* knowing how the are implemented (defined). A function can be used many times within the same program (but must only be defined once!), and often they are *called*, or activated, to operate on different pieces of data. For instance, `sqrt(4.0)` and `sqrt(16.0)` represent two calls to the same function `sqrt()` but each one has a different *argument*, and so will *return* a different value.

Functions are used to group together related and useful statements, and once a function has been written, it can be called many times. This produces smaller programs which take less time to write. It would be tedious if each time you wanted to calculate the square root of a number you had to write a Newton-Raphson loop (a mathematical algorithm for numerically approximating the zero of a function), but by declaring and defining a function called `my_sqrt()`, you can use it by simply typing its name and supply an appropriate argument. Another reason for using functions is to *manage complexity*. It allows programmers to work together on the same project, developing different pieces of code which are then combined to produce the final program. By breaking down a complex data processing task into a number of independent, but related, subtasks, you only have to understand a small part of the problem at any one particular time. Functions are a good method for supporting this type of design methodology. In short, they are extremely *useful*.

12.1.1 Pseudo code

Consider a program that asks for a student's name and determines whether they have passed the course, assuming it has access to the appropriate database. In *pseudo-code*, (or at level 2 in the general problem solving algorithm) this would look like:

read a valid student name
find student's marks
find student's attendance record
calculate student's course mark performance indicators
calculate student's attendance performance indicators
determine whether student has passed the course

Each of the above operations could not be represented by a single statement, but *could* be implemented by a function which is composed of several statements. Here, functions could be used to decompose a complex programming task into a much simpler set of instructions.

12.2 Function Declaration and Definition

The `main()` function has been used in all the examples so far, as whenever a C program is executed, processing starts with the first statement in the `main()` function. C's set of library functions, such as `print()` and `sqrt()` also provide the developer with a wide range of pre-compiled routines. However, for any reasonably complex problem it is necessary to design (declare and define) your own functions, and this will now be discussed.

12.2.1 Function Declaration

It isn't always necessary to declare a function, but it is generally good programming practice to do so, as it provides documentation of the functions that are contained in a particular source (program) file and ensures that their file scope is *global*, see section 12.4.1. A function is declared just like a variable, as its *return type* must be specified as well as a unique name (don't give your function the same name as ones that occur in the standard ANSI C libraries). In addition, however, the *arguments* which are passed to that function when it is activated must also be specified. As an example, the ANSI C library function `sqrt()` is declared as:

```
double sqrt(double x);        /* remember the semi-colon! */
```

and this function accepts an argument `x` of type `double` when it is called and returns a value of type `double` when the function stops (control is passed back to the calling function). A general function declaration can be expressed as:

```
<:data type:> function_name(<:argument list:>);
```

where the return type can be any C data type, as specified in Table 2 (but not an array), and the argument list is composed of a comma separated list of data types with (optional) variable names. This transfer of variables in the calling procedure is illustrated in Figure 15.

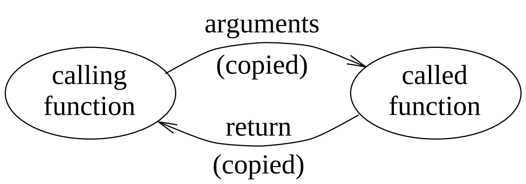

Figure 15: When a function is called, the arguments are copied to the called function, which then returns a value to the calling function after execution has finished.

Functions are often declared at the *beginning* of the source file, *before* the `main()` function, as illustrated in the following code segment:

```
#include <stdio.h>

double my_sqrt(double x);           /* my square root function */
int factorial(int n);               /* factorial (n!) of an integer */
double frac(int num, int denom);    /* two arguments - comma separated */

main()
{
  <:body of main() function:>
}
```

This allows these functions to be called by *any* other function (including itself, see section 12.5) in this source file. Functions can also be declared inside other functions, but this means that they can only be called from within the same function where they were originally declared. The former method for declaring functions globally is the most commonly used.

It isn't always necessary to declare a function; for instance when its definition precedes any calls in the source file, the compiler will accept its definition as the declaration. However, this means that the simplest (and newest) functions are always at the top of the source file and hence it is more difficult to find the most important functions like `main()`. Changing the ordering of the functions in the source file could cause the program to stop working. If however, all the functions are declared at the start of the source file, it overcomes all these problems.

Why is it necessary to declare functions and specify the argument types? Well this allows the compiler to check that the functions are being called correctly and to perform any implicit type conversions that may be necessary. For instance, a function call such as:

```
double x;
x = sqrt(4);
```

could be regarded as being incorrect because an integer `4` is being passed across to a function `sqrt()` whose argument is of type `double`. However, because the function is declared in the header file `math.h`, the compiler recognises that it should convert this `int` into a `double` and as such, the function call is correct. Pre-ANSI C compilers didn't require function arguments to be specified and this caused a lot of problems when programmers passed incorrect argument types to the functions.

12.2.2 Default Declarations

If no return type is specified when a function is declared, the compiler implicitly assumes that it returns an `int`. Similarly, if the function is not declared, the compiler will assume that it returns an `int` and will signal an error if the function is used in any other way. This is why no return type has to be specified whenever you write the `main()` function, but it may return an integer value such as `return 0;` or `return EXIT_SUCCESS;`. Similarly, when no arguments are specified in a function's declaration, the compiler is unable to do any implicit type conversions (such as promoting an `int` to a `double`). As will be shown in section 14.6.1, the `main()` function actually receives two arguments from the calling environment, but because they're not used in our programs, it is not necessary to declare them.

Note that from now on, you should define the `main()` function such that it explicitly returns an `int`. This will be done in all the following examples.

12.2.3 Function Definition

A function definition refers to the piece of C code that states which operations it performs. It occurs *outside* the body of the `main()` (or any other) function and has a general form given by:

```
<:data type:> function_name(<:argument list:>)
{
  <:body of function name containing:>
  <:local variable declarations and statements:>

  return <:expression:>;
```

```
}
```

The first line of the function's definition (data type, name and argument list) should be the same as its declaration; apart from the fact that you are allowed to change the name of the variables in the argument list. The local variables used inside this function are then declared and the statements following this manipulate the local variables, and an appropriate value (possibly nothing) is returned to the calling function.

The group of local (or automatic) variables which can be used inside the function consists of those declared in the argument list as well as any declared at the start of the function. Like the `main()` function, variables must be declared immediately after the opening brace, `{`, and before any other statements.

The function's definition is code which makes up the "black box" represented by the function. It manipulates the local (and global) variables to achieve the desired objective.

12.2.4 Example: `my_sqrt.c`

This may seem a little abstract, so consider defining the function `my_sqrt()`, which is declared as:

```
double my_sqrt(double x);
```

A simple version would be:

```
/* define my_sqrt() which has a double argument variable x and
 * returns the value of a variable of type double
 */
double my_sqrt(double x)
{
  double x_old;          /* declare and initialise local variables */
  double x_orig = x;

  if (x < 0.0)              /* check that x is >= 0 */
    exit(EXIT_FAILURE);            /* if not halt the program */

  if (x == 0.0)             /* check if x == 0.0 */
    return x;

  do {                      /* Newton Raphson calculation */
    x_old = x;
    x = 0.5*(x + x_orig/x);
  } while (fabs((x-x_old)/x) > 0.0001);   /* fabs() is in math.h */
                        /* V. simplistic termination criterion!! */
  return x;             /* return the answer */
}
```

In this function, the variables `x`, `x_old` and `x_orig` get created whenever the function is called and get destroyed when it returns back to the calling function. Therefore, the *local variable set* consists of the three variables `x`, `x_old` and `x_orig`. When the function is called, the value of the expression inside the parentheses in the calling function is copied into the local variable `x`. When this function finishes executing the statements inside its body and the `return` statement is encountered, this value is passed back to the calling function. In this example, the value passed back is stored in the local variable `x`, but in general, the return expression does not have to include any of the arguments. All the statements (declaring, initialising, `do while` loop and arithmetic operations, `return`) lie inside the *function's body*, which is denoted by the opening and closing braces `{ }`, respectively.

A function must either be defined or declared before its first call in the source file. If not, the compiler will signal an error. It is partially a matter of style, whether or not you *declare* all your functions at the top of the source file and then define them after the `main()` function, but this is a style which is consistent with building large programs and will be encouraged in these notes, as stated above.

Functions therefore represent a chunk of computer code which can be executed by simply typing its name, just as the value of a variable can be retrieved by using its name. It can receive arguments in the brackets which come after its name and it can return values (it can even modify its arguments, consider the function `scanf()` which reads a number). Functions are fundamental to good programming practice, and just like meaningful variable names, correct use can simplify a program's design, and reduce its development costs and make it significantly easier to maintain.

Note that the function `exit()` is like a `return` statement in the `main()` function as it signifies to the computer that the program's execution should be halted here. Unlike `return` however, the `exit()` function can be used to halt execution from *any* function. A non-zero argument generally signifies that the program was stopped prematurily, and the macros `EXIT_FAILURE` and `EXIT_SUCCESS`, defined in the header file `stdlib.h`, can be used as its arguments.

12.2.5 `void` Data Type

In the C language, a function must return a data type once it finishes execution. The `main()` function returns an integer value, the function `my_sqrt()`, which is declared above, returns a value of type `double`, but sometimes a function does not have to calculate an answer and its only job it to produce a side-effect (such as printing a message to the screen). In this case, the function must be declared so that it returns a `void`:

```
void print_message(void);
```

In this example, the function is declared as accepting no arguments (`void` argument list) and does not return a value (`void` return type).

12.2.6 Example: `circumference.c`

```
/* This program calls a function to display an initial message
 * and one to calculate the area of a circle.
 */

#include <stdio.h>

#define PI 3.14159265

double circum(double r);      /* these are global function declarations */
void print_title(void);       /* remember the semi-colons! */

main()
{
```

```
  double result, radius;

  print_title();

  printf("Enter radius of circle :");
  scanf("%lf", &radius);

  result = circum(radius);

  printf("The circumference of a circle with radius %6.1f is %8.1f\n",
          radius, result);

  return 0;
}


/* double circum(double r)
 *  This is a function which returns the radius of a circle.
 *
 * Arguments:
 *  double r: circle radius
 */
double circum(double r)
{
  double circumference;

  circumference = 2.0*PI*r;

  return circumference;
}


/* void print_title(void)
 *  This function simply prints text to the screen it has no
 * arguments and returns no value.
 */
void print_title(void)
{
  printf("\nThis program reads a value from the keyboard as the\n");
  printf("radius of a circle and calculates the circumference.\n");

  return;
}
```

12.2.7 Examination of `circumference.c`

- `double circum(double r);`

  This statement declares the function `circum()` to return a variable of type `double` and to be passed an argument of type `double` (as are the variables `result` and `radius`, respectively). A function can be passed and return any data type `int`, `float` etc., and these *must* be declared before both the calling function and the actual function definition to ensure that the compiler processes them correctly. When a function is declared, the variable names do not need to be specified, only their data types, although it is good programming practice to specify meaningful names as well. Indeed it is common practice to use full self-commenting names in the function's declaration and abbreviated names in the function's definition.

- `result = circum(radius);`

  The arguments passed in the function call (i.e. `radius` in `main()`) should have the same data type as the called function was declared to accept (i.e. `r` in `circum()`).

- `double circum(double r)`

  The function *definition* must include a variable name for the arguments as well as their data types. Note that several arguments can be passed by separating the variable names by commas, i.e.

  ```
  double circum(double r, double pi)
  ```

  and this function would receive two arguments. All the code in the braces will be executed (until the first `return` statement) whenever the function is called, and it is also worthwhile remembering that functions can call other (sub)functions, so a *top-down* tree-type structure can be constructed which breaks down the programming task into much simpler blocks.

- `return circumference;`

  The statement `return <:expression:>;` returns control to the *calling* function and also the name of the function evaluates to the value of the expression.

- `void print_title(void)`

  A function declaration preceded by the keyword `void` informs the compiler that the function does not return a value, and the `void` argument list signifies that it does not accept any arguments. This can be useful displaying large messages to the screen, as having all the `printf()` functions in the `main()` function would make it appear overly complex.

- Both functions are declared at the top of the source file and as such can be defined *anywhere* after these declarations. In addition, they can be called by *any* other function.

12.3 Arguments

The calling program communicates with the function by means of its argument list and its return value. For instance, the `sqrt()` function is used to calculate the square root of the variable that occurs between its brackets. The same function may be used to calculate the square root of different variables as all that is required is that the *type* of the arguments must be the same. In the header file `math.h`, the function is declared as `sqrt(double x)` and so long as the variable that is passed across is of type `double`, the function will operate as expected.

Pre-ANSI C compilers did not require the argument types to be specified in the function definition, which was left blank. However, the new style (that requires the data types to be given explicitly) enables the compiler to check that the function is being used properly and also enables built-in type conversion rules to be used. For instance, although the `sqrt()` function is declared to receive a `double` argument, it can be called with the following statement:

```
int number = 4;
double sqrt_ans;

sqrt_ans = sqrt(4);
```

as the compiler "knows" that it should receive a `double` and converts the integer `number` to a type `double`.

Remember that whenever you include the maths library `math.h`, the compiler flag `-lm` must be included when you compile the program.

12.3.1 Call by Value

In C, functions use *call by value* in argument passing. The value of each argument in a function call is assigned to the corresponding parameter in the calling function. In the previous program the variable `radius` in the `main()` function is different from the variable `r` in `circum()`. The local variable `r` is *created* whenever the function is called and its value is set equal to `radius` in the calling program. Therefore, there values are the same but the variables are stored at different memory locations. This means that the variables passed as arguments to functions are *not changed* in the calling environment.

12.3.2 Example: `fn_return.c`

```
#include <stdio.h>

main()
{
  int num = 5;
  int return_num;
  int func(int number);     /* declare function locally */

  printf("Main: num = %d\n", num);
  return_num = func(num);         /* call function */
  printf("Main: num = %d, return_num = %d\n", num, return_num);

  return 0;
}       /* end main() function */


/* int func(int number)
 *   This function increments its own local
 * by two and returns this but this does NOT
 * change the value of the calling argument.
 *
```

```
 *  Arguments:
 * int number: dummy variable
 */
int func(int number)
{
  number += 2;
  printf("Func: number = %d\n", number);

  return number;
}       /* end func() function */
```

12.3.3 Examination of fn_return.c

The output from this program is:

```
Main: num = 5
Func: number = 7
Main: num = 5, return_num = 7
```

When `num` is passed as an argument, `num` is evaluated to produce a value, and this is passed to the function, i.e. passed to the variable `number` in the function `func()`. Thus in the calling environment the variable `num` does not get changed. You could even call the variable in the function `func(int num)` and it would not matter that the names were the same. We will see in section 14 how to change the value of arguments passed to functions, but for the time being, think about how `scanf()` achieves this using addresses.

These variables are *local* to the function that declared them.

12.3.4 Argument Evaluation

When a set of arguments are passed to a function, it is important to notice that their values are evaluated *before* being passed. Consider the following code segment which calls the function `func()`, defined in `fn_return.c`.

```
int num = 5;
int return_num;

return_num = func(num+2);
```

Before the function `func()` is called, all the arguments are evaluated and then their values are passed across to `func()`. Therefore, `func()` is called with a value of 7, and the value of `num` remains unchanged. Hence, at the end of this code segment, the value stored in `num` is 5 and the value of `return_num` is 9.

12.4 Variable Scope

So far, you have not had to worry about spatial and temporal scoping rules for variables and functions. All the variables that are declared at the start of `main()` function can be used anywhere within that function and always exist when the program is running. Functions are meant to partition the programming task into smaller subproblems that have well-defined interfaces. Access to particular variables is controlled via the function's argument list. Any variable that is contained in the calling function and not specified in the argument list is not available to the called function. In `fn_return.c`, the variables `num` and `return_num` are declared in `main()`, but only one is passed to `func()`, and this function has no access to the value held in `return_num` which is "hidden" from it. Variables declared at the start of a function's body are created whenever the function is called and destroyed when control is passed back to the calling function. Such variables are known as *automatic*.

It is common to have variables with the same name defined in different functions, and this is perfectly legal as only one variable will be active at any particular time. For instance, `func()` in `fn_return.c` could have been defined as:

```
int func(int num)
```

This is *not* the same variable as declared in `main()`, although the value will be the same as `num` is the function's argument. This is because the variables in the two functions access different parts of the computer's memory (they have different *addresses*), although when `func()` is called, the variable's value in the argument list is set equal to the integer in the calling function. This process is illustrated in Figure 16.

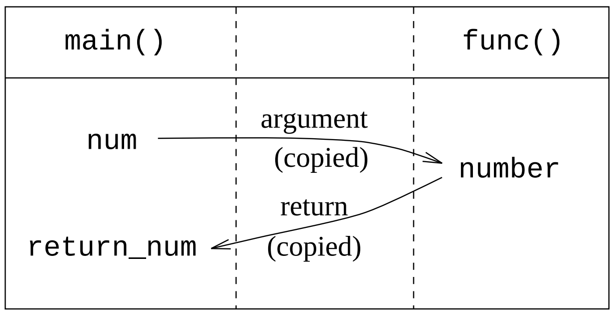

Figure 16: An illustration of what happens when a function gets called.

12.4.1 Global Variables

Global variables and global functions are declared *outside* any other function, and these are accessible by any function *after* they occur in the file. Global variables can be declared and initialised in the same way as normal variables, but their use overrides the normal modular design of functions and as such are discouraged (you'll be penalised for using them unless you can provide a satisfactory explanation). *Constant* global variables are the only real reason for using this feature, and an example of this is:

```
const double my_pi = 3.14159265;

main()
{
  <:body of main function:>
}
```

The `const` keyword which prefixes a variable's declaration means that its value can never be changed during the program's execution. They are similar to the `#define` expressions, except that they cannot be used to set the size of an array (this is one area where C++ is different from C), but they can be seen by a debugger. If you need to declare an array size use `#define`, otherwise use `const double` (or `float` or `int` etc.).

The use of global functions is more common, as when you include a library header file with the `#include` keyword, all of the functions declared in the header file are global. You can make your own global functions by declaring them *before* the `main()` function at the start of the source file, and it is common programming practice to declare most functions as global, unless they will only ever be called from one function.

Other keywords such as `extern`, `register` and `static` can also be used to describe the when and where a variable is created and destroyed. These will be discussed in section 13, where the concept of *file scope* will be introduced and *function libraries* will be discussed.

12.5 Macros and Recursion

To round off this section, we will discuss two aspects of functions associated with the C programming language: macros and recursive functions.

12.5.1 Macros

Conventionally, when a function is called, the set of local variables must be created and initialised for the called function. This obviously takes time for the computer to perform, so it is unwise to write functions which are extremely small if they get called many times, as the *overhead* associated with calling a function may be considerable. *Macros* allow you to implement small operations without the overhead associated with calling a function.

You know that you can declare global constant variables with a `#define` expression such as:

```
#define PI 3.14159265
```

and the C preprocessor replaces every occurrence of `PI` in the program with its equivalent expression `3.14159265`. In this respect, the `#define` construct with the C preprocessor acts like a global find and replace method (a similar function can be found on the menu bar of your text editor). Similar expressions can also be used to implement macros such as:

```
#define MY_SQ(X)  ((X)*(X))
#define MY_MAX(X,Y) (((X) > (Y)) ? (X) : (Y))
```

and instead of a function being called, their equivalent representation is substituted by the preprocessor into the program's code. Using macros can make a program easier to read and understand when the same expression occurs several times in a program (they do not slow down the program as there is no "true" function call), but they should be used with care and the brackets which occur in their definitions are all necessary! Consider why the following definition of `MY_SQ()` is inappropriate:

```
#define MY_SQ(X) X*X
```

A function's arguments are evaluated prior to it being called whereas a macro is simply substituted into the program's code. So in either of the following two cases, the result would not be as expected.

```
ans = MY_SQ(x+1);       /* evaluates to ans = x + 1*x + 1; */
ans = y/MY_SQ(x);       /* evaluates to ans = (y/x)*x; */
```

Note that you can use the debugger to step into a function call and evaluate all of its arguments, local variables etc. However, you cannot evaluate a macro using the debugger as the pre-processor temporarily altered the state of the original source file before compilation. Therefore, errors such as these are difficult to track down.

12.5.2 Recursive Functions

In the C programming language, functions are allowed to recursively call themselves, and this is a common feature in functional-based programming languages such as LISP. A widely used example of why this is desirable is provided by the mathematical definition of the function $n!$, or n factorial. This can be written as:

$$n! = \begin{cases} n*(n-1)! & \text{if } n >= 2 \\ 1 & \text{if } n = 0 \text{ or } 1 \end{cases}$$

for a non-negative integer n. One of the features of the C language is that it can support many programming styles, and the above definition of the factorial function can be written as:

```
int factorial(int n)
{
  if (n >= 2)
    return (n*factorial(n-1));  /* recursion part */
  else if (n >= 0)
    return 1;                   /* terminating condition */
  else
    return 0;                   /* this signifies an error */
}
```

which is very close to the original definition of the mathematical procedure. Recursive functions can be used to simplify otherwise complex expressions, but like macros they should be used sparingly. Recursive functions can slow a program down significantly as they involve calling many very simple functions, and the overheads associated with this are considerable (the memory overhead may in some cases crash the program if too many recursive function calls are made). Similarly, many simple recursive functions can be replaced by `for` loops.

12.6 Summary

- A function is *declared* by specifying its return type, its name and an argument list contained in a set of parentheses followed by a semi-colon. For instance:

  ```
  double frac(int num, int denom);
  ```

  declares a function called `frac()` which receives two integer arguments `num` and `denom` and returns a value of type `double`. Functions are generally declared *globally* at the start of the file, before the `main()` function. A function must also be *defined.*

- A function may return (cease execution in that function) at any point in its body by simply using the `return` keyword, and more than one `return` keyword may occur in a function. A function returns a *single* value to the calling function, or if the function's return data type is `void`, it returns nothing.

- Arguments are passed *by value* and cannot be modified by the called function. Similarly, the called function can only access its own local variables and cannot access any local variables in the calling function.

- Two good rules of thumb are that a functions should be no more than 1 page long and it should not receive more than six arguments (preferably about two or three at most, see the pre-written examples used in C's libraries). This should ensure that a function represents a sufficiently specialised subtask which can be understood easily by another programmer.

- Macros can be used to implement *very short* functions when program running time is critical, and functions are allowed to call themselves in a recursive manner.

13 Function Libraries

Functions are very useful for code *reuse* as they allow the same piece of code to be used many times in the same program or to be called from several different programs. As an example of this, consider the `sqrt()` and `printf()` functions which you have used many times but only have one definition and declaration in the system. By reusing a well-written function which is known to be *bug-free*, a programmer can save time and reduce the chances of making further mistakes. However, this extra flexibility can significantly increase the amount of information that the program designer must initially consider, so in this section we shall look at several methods for dealing with it. In particular, we will investigate how a program can be written in several different files, using *header* and *source* files to create function libraries and archives, and using a *make* file to manage the overall compilation of the project.

13.1 Using header files

Header files allow all the global constant and function declaration to be placed in single file, *separate* from the program or function definition. This separates the function's *public interface* from its actual definition which the user doesn't need to know about. This may initially seem confusing, but consider how you learnt to use the `sqrt()` function: you didn't know how it was defined rather you only knew its interface declaration was declared in `stdio.h`. This separation of the function declaration and definition into two different files allows the designer to *hide complexity* and facilitates the development of larger and more complex programs.

Header files are included at the start of the function definition file using the `#include` command which is followed by the header file's name enclosed in either double quotation marks `" "` or in angled brackets `< >`. The latter denotes an ANSI C standard header file, such as `stdio.h` or `math.h`, and is generally located in the directory `/usr/include` on the Unix system, whereas the former is usually used when the programmer creates their own header file. Examples of this could include the following lines of code at the start of a program (source) file:

```
#include <stdio.h>           /* usually ends in .h */
#include <math.h>            /* ditto */
#include "my_header.h"       /* file in same directory */
#include "include/another.h" /* another.h in subdirectory include */
```

When the header file does not live in the same directory as the program file, either its full or relative pathname should be specified (although it is possible, infact usual, to set this as a compiler option).

The header file itself simply contains global constant and function declarations as well as a couple of "tricks" to ensure that it is included only once. If a header file was included more than once, the compiler would think that the same variable/function was being declared more than once and would flag an error. Hence the `#ifndef` -type commands ensure that the header file is only included once. What really happens is that the contents of a header file get expanded whenever that file is `#include`d by the C pre-compiler. By placing global variables and function declarations in such a file allows them to be shared by several source files.

Including a header file in the function definition file simply means that that header file will be opened at the place that the `#include` statement occurs. Therefore, any statements that occur in the header file must be *valid* C, otherwise the compiler will flag an error.

13.1.1 Example: my_header.h

```
#ifndef MY_HEADER_H     /* has this been included */
#define MY_HEADER_H     /* if not say its been done once */

#define N_MAX   100              /* global constants */
#define PI      3.14159265

#define SQ(X)    ((X)*(X))       /* function macros */

void print_title(void);          /* function declarations */
double sqrt_newton(double init);
void print_results(double appr, double ans);

#endif                  /* end the #ifndef construct */
```

Here the symbolic constant `MY_HEADER_H` which gets declared on the second line should be *specific* to just this header file. If you create more than one header file, choose an appropriate name for the symbolic constant in each file. Generally, this is taken to be the header file's name in uppercase, with an underscore, `_`, replacing the dot, `.`, character.

This header file can be included in a file `my_functions.c` which actually declares the functions, as well as being included in other source files which contain the `main()` function and calls to these functions.

The header file is the interface for the functions defined in the corresponding source file, and should be included in any other source file which calls these functions. Similarly, if you want to use the global constants or structures declared in a header file, it must be included before they are first used.

13.2 File Compilation

One advantage of splitting the program into several separate files is that they can be compiled individually to produce *object* files which can then be *linked* together to produce an executable program. If changes are then made to a single file, it alone has to be recompiled and then linked with all the other object files. This process of splitting the programming task up into several files also allows programmers to work on separate tasks and each person can write and edit their own file. Each file can be compiled individually which will flag the majority of programming errors, and each object file will be linked together and for large programming projects, it is much quicker to link various object files than to compile from "raw" C code.

To produce an object file (`.o`) from a C file (`.c`) you just need to use the `-c` switch on the `cc` compiler. For instance, the command:

```
cc -c my_functions.c
```

would produce an object file `my_functions.o` assuming that there was no bugs in the C file.

To produce an executable file, just compile the program as normal. For instance, the following command links two object files together:

```
cc my_functions.o another_one.o -o my_program
```

and produces an executable file called `my_program`, as illustrated in Figure 17. Unless otherwise directed with the `-c` switch, the `cc` compiler will always try to produce an executable

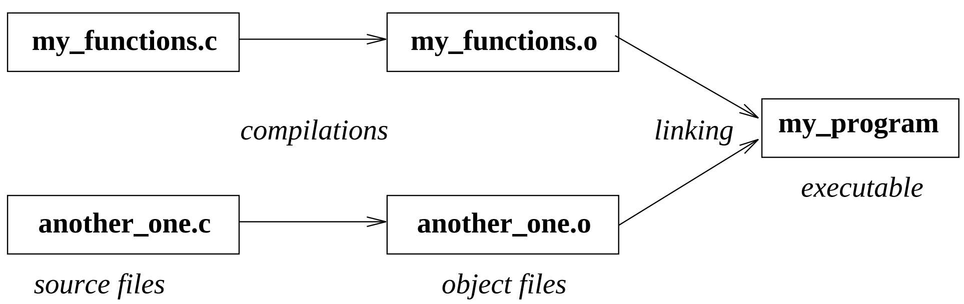

Figure 17: Compiling two C source files to produce two object files which can then be linked together.

file. The `-c` switch stops the compilation process half way (roughly speaking) through. It should be remembered that you can have only *one* `main()` function in a program and so only one of the object files should have a `main()` function in it. This should emphasise the fact that you cannot execute an object file; it must be linked first.

However, all the variables and functions declared in a file are *local* to that file. Their *scope* is only that file, and when a source file is compiled all the variables and functions must be declared. So how can a function be called in one file and be defined in another and both files be compiled separately? This is possible using header files which can be imagined as a file's *interface* with other files. The same header file would be included in *both* files, and when they are compiled separately, the appropriate global functions and variables are correctly declared.

13.2.1 `extern` and `static` variables

It is generally preferable to declare all the variables and functions which are made available to more than one file in the header file, but it *is* possible to access other data and functions in other source files using the `extern` keyword. Global declarations such as:

```
extern int index;
extern double answer;
```

tell the rest of this source file that `index` and `answer` are integers and floating point variables, respectively, but no space is allocated for their storage as it is assumed that they will be declared properly in another source file. If the global variables `index` and `answer` are not declared properly in another source file, the compiler will complain when it attempts to link the object files together.

The `static` keyword allows programmers to declare global variables which can only be accessed from the source file where they are declared. Global declarations such as:

```
static int index;
static void print_message(void);
```

mean that the integer `index` and the function `print_message()` can only be used in the source file where they are declared, and other variables and functions of the *same* name could exist in other source files. This allows a programmer to hide the internal complexity associated with a particular source file from the other source files. It is also possible to declare local variables as `static` and this means that the variables always remain in existence (and hence retain their value) while the program is running, rather than the standard automatic variables which get created and destroyed whenever the function is called.

13.3 Creating library archives

Unix provides a facility called `ar` (an abbreviation for *archive*) that allows object files to be stored in one large object archive, which ends in `.a`. For instance, the archive files for the standard C functions such as `printf()` and `sqrt()` are stored in the directory `/lib` on our Solaris system. To inspect the relevant files, change into this directory and type the two commands:

```
ar t libc.a
ar t libm.a
```

The second command gives you details about the standard mathematics archives whereas the former provides information about all the remaining files. The `libc.a` archive is always linked into your program by default, whereas the mathematics archive file must be explicitly specified using the `-lm` compiler flag.

You can create your own archive files from object files in the following manner. First, it is important to notice that an archive generally consists of several object files, so to create your own library file that consists of two object files, the following commands would be entered:

```
ar rcv my_archive.a first_object.o second_object.o
```

The keys `ruv` simply control the creation of and updating (if necessary) the archive file `my_archive.a`. The archive file can be linked into an executable program just like an object file or a source file, so the following compilation command would take three files (a source, an object and an archive) and produce an executable file called `my_prog`:

```
cc -o my_prog my_main.c my_object.o my_archive.a
```

As you can see the compilation process becomes more complex when a program is split across several files, but the `make` facility can be used to automate this process.

13.4 `makefile` Managing Compilation

The `make` utility can be used to keep track of which files need to be recompiled, based on when they were last edited and compiled, and can perform all the necessary linking. Consider trying to compile the project illustrated in Figure 18 which consists of two libraries (first and second, although generally these would have more meaningful names) and a main source file. As the libraries evolve, it is necessary to keep re-compiling them and to make sure the latest version is linked into the main program. Obviously, as the number of libraries and header files grow, this becomes difficult increasingly more difficult, especially when a header/source file includes other header files and the *order* of compilation becomes important.

To automate this process, the programmer must create file called `makefile`, and this would typically be of the form:

```
my_program: my_main.o  first.o second.o
        cc -o my_program my_main.o first.o second.o

my_main.o: my_main.c first.h second.h
        cc -c my_main.c

first.o: first.c  first.h
        cc -c first.c
```

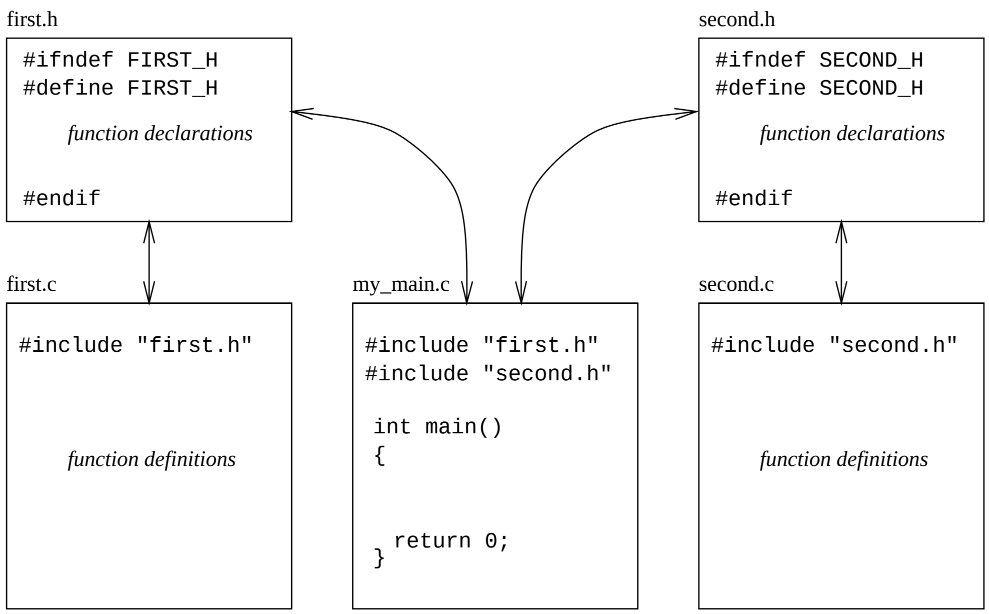

Figure 18: A typical programming project which makes use of two libraries (first and second) and a main source file `my_main.c`.

```
second.o: second.c second.h
        cc -c second.c
```

The file is split into four blocks (separated by a blank line) each of which consist of two lines. In each block the first line (which must start in the first column) has a name followed by a colon and a list of dependencies. This is read as saying that the first name depends on all the others and if any of its dependencies are changed the command on the following line must be executed. The command on the second line *must* start with a tab spacing, and generally the command either creates an object file or else it links several of them together. The compilation process is then initiated by typing `make` (a Unix utility) at the command prompt. If this is the first time that everything has been compiled and linked, it will produce several object files and an executable file, assuming the program was written correctly. However, when some changes are made to only one of the source files, only that file will be recompiled and linked with the other object files. Remembering which files have been edited and keeping track of when you last linked the object files together is greatly simplified using the `make` command.

When a program is split into several source and header files, it is usual to create a separate directory for just those files. Therefore only one `makefile` will exist in each directory and by typing `make` in the appropriate directory will cause the correct compilation sequence to be executed for that program.

13.5 Summary

- Header files (with a `.h` extension) are used to share global variable and functions declarations between different source files (`.c` extension). They should use the `#ifndef`, `#define` and `#endif` pre-processor directives to ensure that they are included only once.

- When a header file is included in a source file, the pre-processor replaces the `#include` line with a complete listing of the header file. Therefore, all of the declarations contained in the header file would then be available in the source file.

- A good rule of thumb is that for each source file which contains function definitions, a header file should be created with the same name (but a different extension - source file `.c` and header file `.h`) which contains the global declarations for that file. This can then be included in other source files if necessary.

- Source files can be compiled individually, using the `-c` compiler switch. This produces a set of *object files* (with a `.o` extension) which can then be *linked* together to form an executable program. Within the set of source files, there should be one and only one `main()` function.

- A library archive (`.a` extension) may be created from several object files.

- Creating a `makefile` for each programming project, which is split into several source files, is extremely useful as it automates the compiling and re-compiling process. Generally, each project would be in a separate directory along with the appropriate `makefile`.

14 Pointers

The concept of a pointer is intricately linked with how ordinary data types are stored in a computer's memory, as they denote the *address*, or location, of a particular variable. The name of a variable determines its type (`char`, `int`, `float` or `double`) and its address determines where it is stored. So declaring a variable such as:

```
int an_int;
```

is equivalent to saying that I wish to reserve 4 bytes of memory at location f7fff840, and these 32 bits should be interpreted as an integer representation (see section 5). Obviously, it is much easier to simply declare a variable! Knowing the address of a variable is important however, as this allows functions to change the values of its arguments (e.g. `scanf()`). It also allows arrays to be passed efficiently between functions as instead of having to copy every entry in the array, only the address of the first element needs to be stored. Finally, and probably most importantly, it allows memory storage to be allocated at *run-time* rather than *compile-time* which means that the size of an array can be determined by the user rather than by the programmer. This makes things slightly more complicated for the programmer, but the potential gains are worth it.

An array name decays into a pointer to its first element (except in three circumstances), and as such the use of arrays and pointers are very closely related. As we have already seen, functions use *call by value*, i.e. values are passed to a function and remain unchanged in the calling function. Pointers are a mechanism for passing arguments to a function when the argument needs to be modified by the function and returned to the calling function. This is known as *call by reference*, see section 14.3.

14.1 Basic Terminology

The value of each variable in a computer program is stored in a section of memory whose size is determined by the variable's data type. The location of this chunk of memory is stored in the variable's *address*, so each variable could be imagined as being composed of two parts: its value and address as shown in Figure 19. Each memory cell can be considered as being

double pi = 3.14159265;

address	value
f7fffa48	3.14159265

Figure 19: A variable in the computer's memory is characterised by its address and its value.

composed of a part that holds its contents and it implicitly has an address. Addresses are usually displayed in hexadecimal form (base 16) whereas the data type stored at the address could be a character, an integer or a floating point number (although in reality all are stored in binary form). To obtain the address of a variable, you *ask* for it with the unitary `&` operator, ie: the program segment:

```
double number = 1.5;
printf("value = %3.1f, address = %p\n", number, &number);
```

would produce an output like:

```
value = 1.5, address = f7fff840
```

since the exact address of a variable depends on the compiler and how and when the program is run. Here, the flag `%p` in the `printf()` function is used to print out the hexadecimal integer representation of an address.

Note that the `&` address operator has already been introduced, and you have used it to modify the arguments in the `scanf()` function.

14.2 Addresses and Pointers

A pointer holds the address of an object in memory and as such a pointer is also a variable. This may initially seem confusing but the address of a variable is just another bit-string whose value can be stored in another variable of a different type: a pointer. Addresses are generally interpreted as hexadecimal integers, so there is no reason why we can't define another data type which is similar to an integer (pointers and integers are *not* interchangeable, they are not even guaranteed to be stored in the same size of memory), but through which it is possible to manipulate the value stored at that address. It is important to clearly understand the relationship between addresses and pointers, and this is summarised in the following two statements:

- Every variable has an address which can be obtained by applying the unary operator `&`.
- The address of a variable can be stored in a data type called a pointer.

Integer pointers are *declared* in the following manner:

```
int *point_to_int;
```

and this is read as saying that the variable `point_to_int` has been declared as an integer pointer, as the unary operator `*` is the inverse of the `&` operator. Pointers to other data types are declared in an analogous manner. A pointer stores an address as its value and applying the unary `*` operator allows the programmer to retrieve the character/number stored at that address. So the following code segment holds:

```
int an_int = 42;          /* declare an integer */
int *point_to_int;        /* declare an integer pointer */

point_to_int = &an_int;   /* store the address of an_int in point_to_int */
*point_to_int = 24;       /* an_int is now 24 */
```

The relationship between pointers, addresses and data types will become clearer in the following examples, but remember that you have already allocated `FILE` pointers in section 6.3 using the unary `*` operator and used the library functions `fopen()` and `fclose()` to allocate, initialise and release memory.

14.2.1 Example: `pointers.c`

```
/* Basic program to illustrate the relationship between pointers
 * and addresses.
 */
```

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

int main()
{
  int *ptr;                   /* declare integer pointer */
  int val1 = 42;
  int val2;

  ptr = &val1;            /* point ptr at val1 */
  val2 = *ptr;            /* store the value at ptr (val1) in val2 */
  printf("val1 = %d, val2 = %d, *ptr = %d", val1, val2, *ptr);
  *ptr=5;                 /* store 5 at *ptr (val1) */
  printf("val1 = %d, val2 = %d, *ptr = %d", val1, val2, *ptr);

  ptr = &*ptr;            /* meaningless but */
  val1 = *&val1;          /* instructive statements */

  return EXIT_SUCCESS;
}
```

14.2.2 Examination of `pointers.c`

When the program is compiled and executed, the output is given by:

```
val1 = 42, val2 = 42, *ptr = 42
val1 = 5, val2 = 42, *ptr = 5
```

and this can be explained as:

- `ptr` is assigned the *address* of `val1`. Therefore the value of `ptr` is the address of the memory location.
- `val2` is assigned the *value* of what `ptr` is pointing to. Therefore, the value of `val2` is the contents of the memory location that `ptr` is pointing to (`val1`).
- After printing out the three values for the first time, the contents of what `ptr` is pointing to is changed to 5. As the contents of what `ptr` is pointing to is `val1`, the value of `val1` is also 5 and `val2` is unchanged.
- The final two statements in the program illustrate how the the `*` and `&` operators are "inverse" concepts, and notice that their order of precedence is right to left, so the one closest to the variable is always applied first.

14.3 Functions: Call by Value and Call by Reference

As we have already seen in section 12, the C language *calls functions by value*, i.e. the value of an argument is passed to a function and the *value* of the variable in the calling function is not changed. But what if we want to change it? This can be achieved by passing the address of the variable which allows the function to know where the variable is and hence change the varaible's value stored at that location. This is known as *call by reference*, and is illustrated in the following example where a function is used to swap the values held by two variables.

14.3.1 Example: swap.c

```
/* Swap two values - wrong way using variable names
 *                 - right way using pointers
 */

#include <stdio.h>
#include <stdlib.h>
                         /* declare functions */
void swap_wrong(int a, int b);
void swap_right(int *aptr, int *bptr);

int main()
{
  int value1 = 1;        /* declare and initialise two integers */
  int value2 = 2;

  printf("start     : value1 = %d, value2 = %d\n", value1, value2);
                         /* try and swap their values the wrong way */
  swap_wrong(value1, value2);
  printf("swap_wrong: value1 = %d, value2 = %d\n", value1, value2);
                         /* try and swap their values the right way */
  swap_right(&value1, &value2);
  printf("swap_right: value1 = %d, value2 = %d\n", value1, value2);

  return EXIT_SUCCESS;
}


/*   A function which (wrongly) attempts to swap the values held
 * in the integer variables a and b
 *
 *   Arguments:
 * int a, int b: two integers values for swapping
 */
void swap_wrong(int a, int b)
{
  int tmp;        /* temporary value to aid swapping process */

  tmp = a;
  a = b;
  b = tmp;

  return;
}


/*   A function which correctly swaps the values of two integers
 * *aptr, *bptr and does so by using pointers!
 *
 *  Arguments:
```

```
 *     int *aptr, int *bptr: two initialised integer pointers
 */
void swap_right(int *aptr, int *bptr)
{
  int tmp;         /* temporary value to aid swapping process */

  tmp = *aptr;
  *aptr = *bptr;
  *bptr = tmp;

  return;
}
```

14.3.2 Examination of swap.c

The output from this program is:

```
start     : value1 = 1, value2 = 2
swap_wrong: value1 = 1, value2 = 2
swap_right: value1 = 2, value2 = 1
```

and the following points should be noted about the program.

- The `swap_wrong()` function has the *values* of `value1` and `value2` passed to `a` and `b`, respectively. Hence the values of `value1` and `value2` in `main()` remain unchanged and `a` and `b` are swapped (using the temporary variable `tmp`). This is lost however, when control is passed back to the `main()` calling function and the automatic variables `a` and `b` are destroyed.

- The `swap_right()` function has the *addresses* (indicated by the `&` symbol) of `value1` and `value2` passed to the *pointers* `aptr` and `bptr`, respectively. It is important to note that the pointer in the argument list is declared as `int *aptr`, but the address of a variable `&value1` is passed across from the calling function. This notation may seem a little strange initially, but it is correct.

It is important to remember to apply the address of, `&`, operator when the functions are called and the corresponding arguments are pointers. Sometimes the compiler may not complain if you forget to do this, although the program will behave incorrectly. Generally it is only possible to find these bugs with a *debugger*.

14.3.3 Modifying Variables and Arguments

Functions can be classified as belonging to either one of two groups: those which modify the (local) variables in the calling function and those which do not. The latter are called for their side effects, such as printing a message to the screen etc., whereas the former are used to modify the calling function's *state* (such as `sqrt()` etc.). You now know of two ways for modifying the variables in the calling function:

- To change a variable in the calling function by having the called function pass back an appropriate value as its `return` type.

- To change variables in the calling function by using pointers to effect a call by reference.

When only a single value needs to be remembered (such as calculating the value of a square root), the result should be passed back as a `return` type, whereas the latter is used when several variables must be altered.

We shall see that sometimes arguments are passed by reference when the data type is large, as instead of having to copy the complete variable, only its address needs to be passed across.

14.4 Pointers and Arrays

In C, an array name often decays into a pointer which references the address of the first element of the array, as illustrated in the following code segment:

```
int arraya[3];
int *aptr;

aptr = arraya;
```

Since `arraya` decays to address of the first element of the array, the above statement is equivalent to:

```
aptr = &arraya[0];
```

and the value of the first element in the array can be referenced as `*aptr` or as `aptr[0]`. Note that you can use array-like notation for pointers to retreive and set the value of the corresponding elements.

It has twice been mentioned that although array names and pointers are similar in many respects, they are not the same. Arrays are used to reference a piece of memory whose size is determined at compile time, therefore you can't assign a new address to the name of an array, ie.

```
arraya = aptr;
```

would be invalid. Differences also occur when the array is the operand of the `sizeof` or `&` operator, or is the string literal initializer for a character array. In practice, these last three differences shouldn't affect you, but you should remember that they do exist.

14.4.1 Array Limits

Before delving too deeply into the relationship between pointers and array names, it is worthwhile examining an obscure property of the way that C deals with arrays. In C, an array could be declared as:

```
int array_int[100];
```

which reserves storage space for 100 integers, starting at `array_int[0]` and finishing at `array_int[99]`. However, a standard C compiler allows you to access the "integers" `array_int[-1]` and `array_int[100]` without complaining! In fact, any index is legal, as the C compiler doesn't necessarily check to see that the number lies within the array's limits (some compilers perform bounds-checking on a program by explicitly setting a flag).

An array is stored as a contiguous piece of memory (in the previous example `100` $\times$ `sizeof(int)` bytes long) and an array retrieves information by looking at the integer stored at `index`×`sizeof(int)` bytes from the start. This index can be positive or negative and can take any value, however, by exceeding the array's limits the likelihood is that the memory is being used for another purpose and if you try and read it, it will produce garbage and if you try to write to it, it may crash the system or overwrite other information without you realising it! So the obvious question is why does C let you do this?

As was mentioned in the introduction, C is flexible in that it lets the programmer to do many things and it "trusts" the programmer to know what they're doing. By allowing you to pass the address of an element in the middle of an array to a pointer, this can then access elements which occur before it using negative indices. This is sometimes a useful feature, but more often than not is can be a major cause of bugs, and a lot of other programming languages actually check the value of the index against the array's limits. Towards the end of this course we'll investigate building array-type data types which store their own array limits and can validate the values entered by the programmer. This is sometimes known as a *fat pointer* where the array bounds are stored together with the pointer to the memory location.

14.4.2 Pointer Arithmetic

Before proceeding further, it is worthwhile looking at what legal operations can perform on a pointer, as a pointer is a data type *similar* to an integer. A small set of operations are defined for pointers:

- The addition or subtraction of an integer to/from a pointer yields a new address.
- Pointers can be compared with one another, using logical expressions, to check whether or not they're pointing to the same piece of memory
- Subtraction of two pointers is a valid operation. The result is the *number of variables* between the two addresses.

and these are illustrated in the following code segment:

```
int arraya[3];
int *startptr = arraya;
int *endptr = &arraya[2];

arraya[2] = 15;       /* print arraya[2] three times using pointers */
printf("%d, %d, %d\n", *(startptr+2), startptr[2], *endptr);

if (startptr == endptr)                   /* compare pointers */
  printf("This shouldn't happen!\n");
                                          /* subtract pointers */
printf("no of elements = %d\n", endptr-startptr+1);
```

There are several points to note about these statements:

- `startptr` is the address of the first element of `arraya` and `endptr` is the address of the last element. `int *startptr = arraya;` is a combined declaration and initialisation statement.
- The expression `*(startptr+2)` increments the value of the pointer `startptr` by two and retrieves the number stored at that location, i.e. it would point to the third memory location and its value is 15. This is equivalent to the expression `startptr[2]` which gets translated to the former expression by the compiler! Generally, the array-like notation is preferable as it is more readable.

- The expression `startptr == endptr` checks to see whether the two pointers are equal. This is only true if they both point to the same variable. This is equivalent to deciding whether two arrays are the same, as the array names decay into pointers and the value of the pointers are checked to see if they both reference the same piece of memory.

Whenever an arithmetic operation is performed on a pointer by adding/subtracting an integer, the pointer is incremented/decremented an appropriate number of places such that the new value pointed to is n *elements* (not bytes) ahead/behind the old one. This has obvious parallels with arrays as will be described in the next section. Similarly, subtracting two pointers produces the number of objects between these two locations and gives a measure of how far apart are the two pointers. Finally, two pointers are equal iff (if and only if) they point to the same variable (the value of the addresses are the same). They are not necessarily equal if their indirected values are (as these variables could be stored in different memory locations).

This implies that pointer and array subscript notation are almost equivalent, as illustrated in the following example.

14.4.3 Example: `point_array.c`

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

#define MAX_SIZE 5

int main()
{
                          /* declare array, pointer and end pointer */
  int i, x[MAX_SIZE];
  int *xptr;
  int *end = &x[MAX_SIZE-1];
                       /* subscripts used to initialise the array x */
  for (i = 0; i < MAX_SIZE; i++)
    x[i] = 5*i;
                        /* pointers used to access the same array x */
  for (xptr = x; xptr <= end; xptr++) {
    i = xptr - x;                        /* get current array index */
    printf("x[%d] = %d, %d\n", i, *xptr, x[i]);
  }

  return EXIT_SUCCESS;
}
```

14.4.4 Examination of `point_array.c`

The output from this program is as follows:

```
x[0] = 0, 0
x[1] = 5, 5
x[2] = 10, 10
x[3] = 15, 15
x[4] = 20, 20
```

A more commonly used way of referencing the elements in an array through a pointer is as follows:

```
int i, x[MAX_SIZE];
int *xptr = x;

for (i = 0; i < MAX_SIZE; i++)
  printf("x[%d] = %d\n", i, xptr[i]);
```

As when a program is compiled, it translates `x[i]` (its array representation) to `*(x+i)` (its pointer representation). This is the form you should generally use.

14.4.5 Passing Arrays to Functions

The contents of an array *can* be changed when passing an array name as an argument in a function call because the name of an array decays to the address of its first member or element. That is, arrays are *always* passed by reference.

14.4.6 Example: `array_fn.c`

```
/* Pass an array to a function which assigns the values.
 * Uses (simple) indexing arithmetic.
 */

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

#define MAX_SIZE 15
                                  /* declare fn which takes array argument */
void set_vector(double dbl_arr[]);

int main()
{
  int i;
  double y[MAX_SIZe];

  set_vector(y);                     /* call function with array name */
  for (i = 0; i <= MAX_SIZE-1; i++)
    printf("y[%d] = %lf\n", i, y[i]);

  return EXIT_SUCCESS;
}


/* Set elements in a 1D array (vector) using array arithmetic
 * Don't need to set the array size in argument list.
 */
void set_vector(double dbl_arr[])
{
  int i;
  int num;
```

```
  for (i = 0, num = 1; i < MAX_SIZE; i++, num++)
    dbl_arr[i] = num/3.0;                  /* array is passed by reference */

  return;
}
```

14.4.7 Examination of `array_fn.c`

This example demonstrates that elements of an array can be easily modified in a function and that these changes are reflected in the calling function. Some points to note are:

- `MAX_SIZE` can be legitimately used in the function `set_vector()` as it is a global pre-processor directive and as such every occurrence of the word `MAX_SIZE` is replaced by the value 15 before full compilation takes place. It is quite obvious here that the use of `MAX_SIZE` greatly simplifies making any changes in array sizes, i.e. the program can easily be changed to deal with an array of size, say 50, by modifying just one pre-processor directive.

- In the function declaration:

  ```
  void set_vector(double dbl_arr[]);
  ```

  the size of the 1 dimensional array does *not* need to be specified, as memory has already been reserved for the array and all that needs to be passed is the *address*. Note that this is *not* true for 2 and higher dimensional arrays.

- The `for` loop in the function `load_vector()` steps through the array `dbl_arr`. Note that in this `for` loop, two items are incremented.

- As the array name `y` degrades to a pointer to the first element of the array then, in fact, the address of `y[0]` is passed to the function and *not* its value.

Note that the function `set_vector()` could have been declared and defined as:

```
void set_vector(double *dbl_ptr);
```

and the function would be called in *exactly* the same manner. Indeed the body of the function could be the same (apart from the obvious variable name change), or it could be implemented using pointer arithmetic. However, declaring and defining it to receive an array argument is the simplest and clearest action, as the argument *is* actually an array!

14.5 Multi-Dimensional Arrays

In the C language, a 2-dimensional array is simply a 1-dimensional array whose elements are 1-dimensional arrays and this is illustrated in Figure 20. This representation means that arrays of *any* dimension can be allocated, and the only limit is the amount of memory available on the system.

Due to the relationship between arrays and pointers (and the manner in which arrays are stored) there are numerous ways to access elements of a two-dimensional array. For example, consider the following declaration:

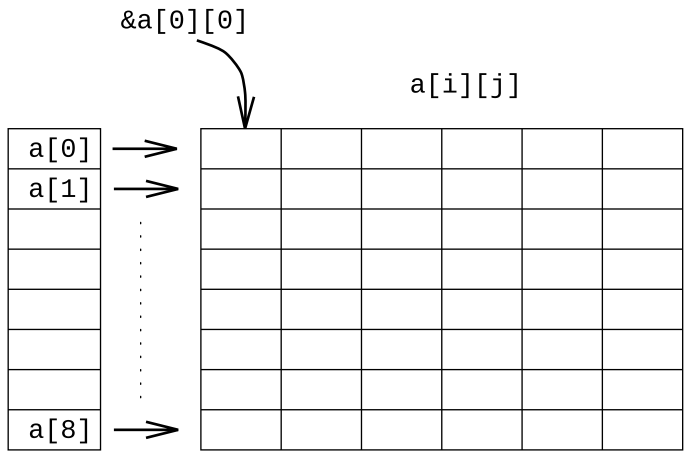

Figure 20: A 2-dimensional array stored as a 1-dimensional array of 1-dimensional arrays.

```
int a[3][5];
```

which declares an array `a` with 3 rows and 5 columns. Then the following expressions are all equivalent to referencing `a[i][j]`:

```
*(a[i]+j)
(*(a+i))[j]
*(*(a+i)+j)
```

as `a[i]` (or equivalently `*(a+i)`) can be regarded as a pointer to the i^{th} row of `a`.

Passing 2-dimensional arrays as arguments is done as shown in the following example.

14.5.1 Example: add_array.c

```
#include <stdio.h>

#define ROW_MAX   3
#define COL_MAX   5
                 /* declare function: size of second quantity MUST be set */
void add_1(double arr2[][COL_MAX]);

int main()
{
  int i, j;
  double temp, array2[ROW_MAX][COL_MAX];     /* declare 2D array */

  printf("Enter elements of 2D array of size %d by %d\n", ROW_MAX, COL_MAX);
  for (i = 0; i < ROW_MAX; i++)
    for (j = 0; j < COL_MAX; j++)
      scanf("%lf", &array2[i][j]);           /* initialise 2D array */

  add_1(array2);                             /* modify elements */

  printf("The new first element is:  %f\n", array2[0][0]);
```

```
  return EXIT_SUCCESS;
}


/* Add 1.0 to each element of the 2-dimensional array
 *
 */
void add_1(double arr2[][COL_MAX])
{
  int i, j;

  for (i = 0; i < ROW_MAX; i++)
    for (j = 0; j < COL_MAX; j++)
      arr2[i][j] += 1.0;             /* modify elements in arr2[][] */

  return;
}
```

14.5.2 Examination of `add_array.c`

The number of columns *always* needs to be specified in the function declaration and definition (whereas the number of rows do not), and this could also be written as:

```
void add_1(double arr2[ROW_MAX][COL_MAX])
```

or

```
void add_1(double (*arr2)[COL_MAX])
```

This last declaration shows how the 2-dimensional array is stored in the computer's memory: as a 1-dimensional array of pointers to `double`s.

In general, only the first set of brackets can be left empty in a multi-dimensional array, all the remaining dimensions *must* be specified.

14.6 Pointers and Strings

To understand how C stores character strings properly, it is necessary to have a good understanding of how it uses pointers. When you read a string into a character string, you should do something like:

```
char name[30+1];
                  /* read in a string (1D character array) called name */
scanf("%s", name);
```

In this code segment, the character array `name` decays to the address of the first character in the string, hence `scanf()` just puts characters into successive memory locations until the end of string marker (white space etc.) is reached. You don't have to specify brackets or indices as the name of an array and the address of the first element are synonymous. In addition, when `scanf()`, reads in a string it automatically `NULL` terminates it for you, so you need to declare one extra character for this. In the code segment above, at most 30 characters could be stored in the string name. If the actual string was longer than 31 characters, your program would probably crash (or even worse keep running with incorrect variable values) as system specific memory may be overwritten.

Choosing the length of a string (or a numerical vector/array) at design-time rather than run-time is undesirable for many reasons and shortly we shall look at how memory can be requested while the program is running to ensure that the correct string length is always used.

14.6.1 The main() Function's Arguments

In general, when a function is defined with an empty argument list, it means that no variables will be passed across to the function. However, when a function is *declared* with an empty argument, the compiler isn't informed about the number and type of arguments, but the function must still be defined and used in a consistent manner. The reason for allowing functions to be declared with an empty argument list is so that ANSI C compilers would still work with old K&R-style C code. In order to declare that no arguments are passed to a function, the void keyword should be used.

However, the main() function is special. In general, it receives two arguments from the calling environment: an integer and a pointer to an array of character strings. They are traditionally called argc (argument count) and argv (argument vector), respectively, and the main() function would be fully defined as:

```
int main(int argc, char *argv[])
{
  <:body of main() function:>

  return EXIT_SUCCESS;
}
```

So far, you've been defining the main() function like:

```
int main()
{
```

specifying no arguments inside the parentheses. In fact, both forms are allowed (but only for the main() function), and if you need to communicate with the program being executed from the command line, the former definition should be used.

The integer argument, argc, stores the number of strings on the command line when the program was executed and the vector of strings, argv, stores their actual values. So if a program called echo_args was executed with the following command:

```
echo_args  these_are  some_strings and_numbers 1
```

argc would have the value 5, and the 5 string pointers (argv[0] - argv[4]) would point to the strings:

```
echo_args\0
these_are\0
some_strings\0
and_numbers\0
1\0
```

where the NULL character, '\0', has been appended to the end of each string. The first string is always the program name and the remaining command line arguments allow users to specify different options when the program is executed, as illustrated in the following example.

14.6.2 Example: echo_args.c

```
/* This program prints the third, fourth, ... command line arguments
 * when the second string is "print".
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>         /* header file for string compare functions */

int main(int argc, char *argv[])
{
  int i;
              /* if the second command line string is "print" */
  if (strcmp(argv[1], "print") == 0)
    for (i = 2; i < argc; i++)    /* print out 2, ..., argc-1 */
      printf("%s ", argv[i]);

  return EXIT_SUCCESS;
}
```

14.7 printf() and scanf() Revisited

It should now be clear why the formats of the printf() and scanf() statements are as they are. These are just functions themselves, albeit system defined functions. Arguments are passed to printf() and can be passed by value. For the scanf() function, the arguments have to be altered and must therefore be passed by reference, hence the use of the & operator to pass the address across.

14.8 Summary

- A pointer is a variable which can store the address of another variable, ie:

  ```
  double *ptr;
  ```

  declares ptr to be a pointer which can store the addresses of another variable of type double.

- The unary address, &, and indirection, *, operators are applied to variables and pointers, respectively, ie.

  ```
  int x, *xptr;

  xptr = &x;     /* store the address of x in the pointer xptr */
  *xptr = 42;    /* change the value stored at the address xptr to 42 */
  ```

- Pointers are used to effect *call by reference* where the called function can modify the value of the argument in the calling function. In C, arrays are always passed by reference.
- Pointers and array names are closely related. In many circumstances, the array name decays to the adddress of the first element in the array. However, they are not equivalent, with the most important difference being that you can't modify location of memory of the array, whereas you can change the value of a pointer.
- When a pointer stores the address of an array (or has had its memory dynamically allocated, see the next section), it is possible, in fact recommended, to use the array-like syntax to access the relevant variables.

15 Dynamic Memory Allocation

In the previous section, pointers were used to effect a *call by reference* which allows functions to change the value of their arguments. This section shows how pointers can be used as a means for allocating memory at run-time rather than compile-time; a technique known as *dynamic memory allocation*. It allows a program to request memory while the program is running, rather than when it is compiled.

C is a *strongly-typed* language in that the type and number of the the variables is known at compile time. This allows programs to run fast and in a predictable fashion. Weakly-typed languages (such as Smalltalk etc.) allow both a variable's type and the number of such variables to be determined at run-time. This is obviously more flexible but it also means that the program has to interpret correctly the information provided by the programmer, which in turn implies that the program will run slower. One of the features that makes C attractive for programmers is that it is flexible, and although it is classed as a strongly typed language, you can use library routines to determine the size of an array at run-time.

This is a frequently used feature. For instance, imagine you were developing a database which stores a record of all the students at Southampton. It would contain their name, course, personal tutor etc., and this would require arrays to be allocated when the number of students is known *not* when the program is written. It may be possible to estimate an upper bound for the size of the array, but this would not be desirable for two reasons:

- The program would reserve more memory than is actually needed, and with todays multi-tasking operating systems that allow several programs to operate at the same time, this could restrict how many tasks could be initiated.
- Any estimate of an upper bound would probably be valid only for a certain number of years due to the inherent expansion of educational facilities. Hard wiring this information into the program at compile time makes it difficult to change, and the program could not be extended easily to another university which had twice as many students.

Dynamic memory allocation is therefore a very useful feature which is quite simple to implement. However, the downside is that the computer may decide that the required memory cannot be allocated at run-time, so checks must be made to ensure that the operation was correctly carried out (see section 6.5).

15.1 `malloc()` and `free()`

Allocating memory dynamically is a fairly simple process which is done by declaring an appropriate pointer (start of an array) and then performing a simple call to the function `malloc()`, which is short for `m`emory `alloc`ation. Once you have finished with the memory, it must be released with a call to `free()`, and both of these functions are declared in the standard library `<stdlib.h>`. The following program illustrates how an array (a variable can be considered to be an array with a single member) can be allocated dynamically.

15.1.1 Example: `memory_alloc.c`

```
/* Dynamically allocate and delete an array using pointers.
 *
 */

#include <stdio.h>
#include <stdlib.h>     /* include for malloc() */
#include <assert.h>     /* include for assert() */
```

```
int main()
{
  int i, array_size;
  int *ptr_array;
                                              /* read in array size */
  printf("Enter array size:");
  scanf("%d", &array_size);
                                         /* allocate memory for an  */
                                /* integer array of size array_size */
  ptr_array = (int *) malloc(array_size*sizeof(int));
  assert(ptr_array != NULL);                        /* check memory */
                                             /* use array like normal! */
  for (i = 0; i < array_size; i++)
    ptr_array[i] = i;
                           /* free the dynamically allocated memory */
  free(ptr_array);

  return EXIT_SUCCESS;
}
```

Here `array_size` is an integer variable which is read in while the program is running and a piece of memory is reserved to hold `array_size` variables of type `int`. The location of the first element is pointed to by `ptr_array`, and the elements can be accessed using the normal array-type notation.

15.1.2 The `void *` Pointer and the `assert()` Function

A pointer must be declared in the program and a number of bytes (`array_size*sizeof(int)` - remember the `sizeof()` operator returns the size of the data type in bytes) are reserved for this array by the `malloc()` function. This memory must be *deallocated* when you have finished using it and this is achieved with a call to the function `free()`, for unlike normal automatic variables, the memory is not freed upon leaving a function. Once a pointer has had some memory allocated to it, it can be used just like a normal array.

One slight variation which you may not have come across before is that `malloc()` is declared as a function that returns a pointer of type `void *`. Declaring a function that returns `void` means that it doesn't pass back any information to the calling environment. However, by declaring that a function passes back a `void *` means that it can return *any* type of address, and this must be *cast* to the appropriate type before it is assigned to a pointer. Hence the `(int *)` cast that occurs between the assignment operator, `=`, and the `malloc()` function calls means that the address will be stored in an integer pointer. Memory for an array of type `double` would be dynamically allocated as:

```
double *ptr_vector;

ptr_vector = (double *) malloc(10*sizeof(double));
assert(ptr_vector != NULL);
```

Remember that you sometimes have to cast an `int` to a `double` in certain mathematical expressions and this is equivalent to the above casting operations. If the computer is unable to allocate any memory at run-time, it will return `NULL` and the value of `ptr_vector` will be `NULL`. This can be used to check whether or not the operation was performed correctly and the useful function `assert()` is often used, especially during initial program design. When the argument expression is false, `assert()` halts the program, just like `exit()`, and prints out where and why it stopped, and if it is true nothing is altered. A more intelligent solution though would be to allow the user some control over how the program runs even when something unexpected like this occurs.

It could be argued that dynamically allocating memory is unnecessary so long as you define an array which is large enough to handle every conceivable situation. However, this is inefficient when only a small part of the array is actually used and the program may not even run when memory is severely limited. Dynamically allocating memory can have its own problems though, which are mainly due to sloppy programming. Once a piece of memory has been allocated, it will not be used for other variables, even if the pointer it was allocated to is now addressing a different part of the memory. Programmers must keep a careful track of which arrays are dynamically allocated and ensure that it is deallocated (free'd) as soon as they are no longer needed.

15.2 Using Pointers as Return-Types and Arguments

Memory allocation and pointers can be used to design well-structured C code, as they allow programmers to write functions that allocate memory and initialise the corresponding variables, all of which is hidden from the calling function. The only type of information that needs to be returned is the address of the memory location where the variables are stored. This is illustrated in the following example which declared and initialises a vector from a file.

15.2.1 Example: `vector_init.c`

```
/* Call a function to initialise a double array from a file.
 * The file contains an integer which specifies the number of
 * elements in the array and then has the that number of
 * floating point values.
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
                 /* declare function to allocate and initialise */
                 /* a double vector from a file. Returns a pointer! */
double *d_vector(char *filename, int *length)

int main()
{
  int i, length;
  double *ptr_vector;
  char filename[] = "vec_info.txt";
                   /* allocate and initialise a double vector from a file */
  ptr_vector = d_vector(filename, &length);
                                      /* print out vector */
  for (i = 0; i < length; i++)
    printf("%f\n", ptr_vector[i]);
                                      /* free memory */
  free(ptr_vector);
```

```
  return EXIT_SUCCESS;
}


/* Allocate and read in a double vector from a file
 * Return a pointer to the allocated memory and store the
 * size at the address pointed to by length.
 */
double *d_vector(char *filename, int *length)
{
  int i;
  double *ptr_vector;
  FILE *read;
                                  /* open file */
  read = fopen(filename, "r");
  assert(read != NULL);
                                 /* read in the length of the vector */
  fscanf(read, "%d", length);    /* length is a pointer so no & needed */
  assert(*length >= 0);
                                 /* allocate memory for vector */
  ptr_vector = (double *) malloc(*length * sizeof(double));
  assert(ptr_vector != NULL);
                                 /* read in the values */
  for (i = 0; i < *length; i++)
    fscanf(read, "%lf", &ptr_vector[i]);

  fclose(read);

  return ptr_vector;
}
```

15.2.2 Examination of `vector_init.c`

By designing a function such as `d_vector()`, a modular design has been adopted which removes unnecessary complexity in the `main()` function. The user supplies a file name and the function reads in the number of elements in the array from this source, allocates an array which can hold the correct number of variables and reads a set of numbers into the relevant locations. The programmer who calls this function does not need to know about file pointers etc., as this is *encapsulated* in the definition of `d_vector()`. This function could be used many times by many different functions, and so it has been declared as a *global* function. It returns the address of the allocated memory (a pointer to a `double`) and it modifies the argument `length` (called by reference) which stores the number of elements in the vector. The following points should be noted about the definition of `d_vector()`.

- You must return the address of the allocated memory and *not* pass it across as an argument which can be modified. This is because if you pass across a pointer as an argument, `malloc()` stores the address of the allocated memory in a copy of this pointer, which will be "lost" when control is passed back to the calling function. You must *copy* the address of the allocated memory and this can only be done by passing it back as the return type. Notice that file pointers and the functions `fopen()` and `fclose()` interact in a similar manner to this program.

- It is necessary to store the length of the vector in a separate number, as you cannot otherwise find out the length of the vector. This is passed across as an argument that can be modified, as it is passed by reference.

For large, complex programs it is common to have separate initialisation, and deallocating functions and sometimes the former is as large as the main body of the program. In the next section, we shall see how to tie together the length of a vector and its pointer into a "new" *user-defined* data type, and associated with this should be separate initialisation and deallocating functions. This has been emphasised in C++ which greatly expands the ideas of a user-defined data type.

15.3 Pointers to Pointers to ...

So far we've seen how to declare a vector at run-time using pointers and the functions `malloc()` and `free()`. To declare a 2 dimensional array is both similar *and* different as shown in the following code segment.

15.3.1 Example: `array2D.c`

```
/* Dynamically allocate a 2-dimensional max_rows*max_cols array
 * of doubles
 */

#include <stdlib.h>
#include <assert.h>

int main()
{
  int i, j;
  int max_rows, max_cols;
  double **ptr;    /* pointer to a 2D array */

  printf("Enter the number of rows and columns:\n");
  scanf("%d %d", &max_rows, &max_columns);

                   /* allocate vector of max_rows double pointers */
  ptr = (double **) malloc(max_rows*sizeof(double *));
  assert(ptr != NULL);                    /* memory allocation OK? */
              /* allocate max_rows rows of arrays of max_cols doubles */
  for (i = 0; i < max_rows; i++) {
    ptr[i] = (double *) malloc(max_cols*sizeof(double));
    assert(ptr[i] != NULL);        /* memory allocation OK? */
  }
                                   /* read in values to matrix */
  printf("Enter the elements:\n");
  for (i = 0; i < max_rows; i++)
    for (j = 0; j < max_cols; j++)
      scanf("%lf", &ptr[i][j]);

  for (i = 0; i < max_rows; i++)   /* free memory (in reverse order) */
    free(ptr[i]);                  /* each row of max_cols doubles */
  free(ptr);                       /* vector of double pointers */
```

```
  return EXIT_SUCCESS;
}
```

15.3.2 Examination of `array2D.c`

C stores a 2D array as a vector of 1 dimensional vectors, therefore in order to be able to allocate memory for a 2D dimensional array, you must define *a pointer to a pointer to* the appropriate data type (denoted by a double star). The first call to `malloc()` allocates memory for `max_rows` pointers to `double`, where each pointer points to a row of the array, and the address of this vector of is stored in the pointer `ptr`. Memory must then be allocated to each pointer in the vector and this is performed using the `for` loop which repeatedly calls the `malloc()` function. Each call reserves storage space for `max_cols` variables of type `double`, which correspond to the number of variables in each row of the 2D array. When freeing up the memory, the rows should be deallocated before the 2D array pointer is freed, i.e. the operations should be performed in the opposite order.

This may seem a little confusing at first, but it is flexible in that it makes it possible to define *ragged* arrays very easily, and this allocation procedure generalises to higher order arrays by defining triple star pointers etc.

15.4 Summary

- Dynamic memory allocation allows the size of an array to be determined at run-time rather than compile-time. This is done by declaring a pointer to store the address of the memory segment returned by `malloc()` and then using `free()` to explicitly deallocate the memory once it is no longer required.

- Once allocated, array-type notation can be used to reference the individual elements inside the dynamically allocated memory.

- Memory can be allocated inside a function and returned to the calling function, by declaring it as a function which returns a pointer. This memory would then be deallocted by the calling function (or another function) when appropriate.

- Multi-dimensional arrays can be dynamically allocated by declaring an appropriate pointer and performing multi-dimensional memory allocation and deallocation.

16 Structures

In section 10 you learned how to construct and use *arrays* which are compound data types, i.e. an array is composed of one or more variables (members), and in an array each variable has the same type. This allows related variables to be grouped together and simplifies the program's structure as it simplifies both both declaring and using such variables.

Structures allow the program designer to group together variables of *different* types and to create *user-defined* data types that closely resemble real-world objects. This is sometimes called *data abstraction* as the program designer concentrates on how the data should be organised. Functions are used to decompose the program's complexity in terms of the operations on the data, whereas structures are used to abstract relevant concepts. This will be emphasised in the next section when we will discuss software engineering in greater detail.

16.1 Declaring a Structure

A structure is a generalised, user-defined data type that can be used to group related pieces of data into a new concept. For instance, a structure that stored records about each member of this class, would be called `Student`, and declared as follows:

```
#define NO_ASSN           2          /* number of assignments */
#define NO_OF_STUDENTS    76         /* no of students in class */

struct Student {
  char *name;
  unsigned short int age;
  unsigned short int assn_marks[NO_ASSN];
  unsigned short int no_lectures_missed;
  double bank_balance;
};
```

where it is a common convention that the name of a structure begins with a capital letter. Given this definition, variables of type `Student` could be defined as follows:

```
struct Student temp, year96[NO_OF_STUDENTS];
```

This declares a single variable `temp` of type `Student` as well as an *array* (`year96`) of `Students`. Each `Student` variable has a `name`, an `age` etc., so instead of having to define separate arrays for each member, you just have to define a single array of type `Student`.

If you find that you're declaring and using arrays like:

```
int age[NO_OF_STUDENTS], assn_marks[NO_OF_STUDENTS][NO_ASSN],
    no_lectures_missed[NO_OF_STUDENTS];

for (i = 0; i < NO_OF_STUDENTS; i++) {
  age[i] = 18;
  assn_marks[i][0] = 0;
  assn_marks[i][1] = 0;
  no_lectures_missed = 0;
}
```

you should be using structures to *group together the related data*. In addition, if you find that you're passing or returning a large number of related variables to other functions, it may be worthwhile grouping them together into a structure.

When you declare a variable of type `Student`, you are hiding a lot of detail about its underlying structure. This is analogous to when you are using files, you don't know how the `FILE` structure is implemented, but you can declare variables of that type and can manipulate is members using functions such as `fopen()`, `fscanf()`, `fclose()`, etc. The concept of a file is understandable, whereas its underlying implementation is very complex, and this is the principle behind *data abstraction*.

16.1.1 File Positioning

Before a structure variable, such as `temp`, can be declared, the structure must be defined in the file. It is usual to define *global* common structures in the header file, as this means that whenever you include the header file into the source file, you can declare variables of that type (note that `FILE` is defined in the header file `stdio.h`) anywhere in that file. Remember, the header file acts as an *interface* between its own source file and any other source file, so if you want to make the definition of the structure public to other files, you must include it in the associated header file.

However, it is possible to define and use *local* structures that are only accessible by one function, for instance. By defining the structure and declaring variables at the same time at the start of the function, you make its definition *local* to that particular function and no other function would understand the structure's meaning. An example of this would be of the form:

```
void my_function(int day, double money)
{
  struct Student {                    /* Student is local structure */
    <:list of structure's members:>
  } temp, year96[NO_OF_STUDENTS];     /* These are local Student variables */

  <:body of my_function(): one or more statements:>
}
```

which would define a structure of type `struct Student` which is local to the function `my_function()` and declare the variables `temp` and `year96[]`. Structures are generally used to pass information between related functions, therefore their declaration is usually global.

16.1.2 `typedef`

The `typedef` command which can be used to re-name `enum` variables, see section 9.2.2, can also be used with the `struct` keyword. For a structure (or any other variable), the `typedef` command must come after it is defined and *before* any variables of that type are declared. In fact, it is common practice to rename the structure when it is defined in the following manner:

```
typedef struct Student {
  <:list of structure's members:>
} Student;
```

This defines a data type `struct Student` as well as renaming it as `Student` all in one go. This is the form you should generally use.

16.2 Accessing Members

In order to access information contained inside a structure, a variable of that type must be declared with appropriate the member variable. This can be illustrated by using the `Student` structure previously declared. In order to access the age of a particular student, we have to specify which student (variable) we wish to access the age of, as well as specifying the characteristic age, in this case. This is achieved using the `.` (dot) operator as illustrated in the following code:

```
Student temp;
                              /* add 1 for terminating NULL character */
temp.name = (char *) malloc((strlen("Fred Bloggs")+1)*sizeof(char));
assert(temp.name != NULL);
strcpy(temp.name, "Fred Bloggs");     /* assign Fred Bloggs to name */

temp.age  = 18;
temp.bank_balance  = -100.0;
temp.assn_marks[0] = 78;
temp.assn_marks[1] = 69;
temp.no_lectures_missed  = 0;
```

Once the `temp` variable has been declared, the information contained in the structure like `temp.age`, `temp.name`, etc., can be initialised, retreived, set, etc., just like any other variable of the same type. Therefore, it follows that when you declare a pointer as a member of a structure, you must allocate some memory to that pointer before you can store information in that location. Similarly, when a variable of type `Student` goes out of scope, that memory must explicitly be deallocated with a call to `free()`, as only the memory used to store the pointer is freed by the compiler. Finally, the members inside an array of type `Student`, would be used in a similar manner:

```
year96[7].age = 18;
for (i = 0; i < NO_OF_STUDENTS; i++)
  year96[i].no_lectures_missed = 0;          /* conscientious! */
```

The name of the member inside each structure variable (`age` etc.) is the same, but because the variables, `year[0]`, `year[1]`, . . . , have different names, there is no conflict.

16.2.1 A Structure inside a Structure

Structures can have as members other structures or pointers to other datatypes, and very large tree-structured hierarchies can be constructed, although this is only useful if it has a well-defined, natural interpretation; such as the concept of a student. For instance, a university has one or more students, so a structure of type `University` could be defined as follows:

```
typedef struct University {
  char *name;
  double income;
  Student student[NO_OF_STUDENTS];
} University;
```

and a variable declared as:

```
University southampton;
```

Members of this structure can then be accessed as illustrated in the following code segment:

```
southampton.income = 121657842.34;
southampton.student[3421].age = 20;
```

It doesn't matter that we've declared the same member `*name` inside both `Student` and `University`, because the structure hides its members away, and the members can *only* be accessed though the appropriate variable. Therefore, `southampton.name` and `southampton.student[23].name` refer to character pointers which access the university's name and the name of the 24^{th} student, respectively.

This idea can be extended still further, and you can have many levels of data abstraction, but it should be remembered that each structure should have an obvious meaning (and this should be reflected in its name), and grouping variables together into meaningless structures would cause more confusion than not using structures at all.

16.3 Copying and Assigning Structures

Two operations that the compiler supports for structures is *copying* and *assignment*. It would be unreasonable to expect any more help as a structure is a completely general data type. However, the actions of copying and assignment allows the designer to initialise a structure using the braces (like arrays) and to set one structure equal to another of the same type. This is illustrated for the following definition of a complex number data type:

```
typedef struct Complex {
  double real;
  double im;
} Complex;
```

where values are assigned to members of `x` and then copied to `y`.

```
Complex x = {3.0, 5.0};    /* assignment x = 3 + 5i */
Complex y;

y = x;                     /* copy y = 3 + 5i */
```

The assignment and copying operations are done on a member by member basis, where the appropriate value is assigned to each member of that structure.

This is an extremely useful feature as *small* structures can be *passed to* and *returned from* functions. For instance, it is possible to create a function `add()` which takes two complex numbers as arguments and returns their sum, and in the calling function this would look something like:

```
Complex x = {3.0, 5.0};     /* x = 3 + 5i */
Complex y = {-1.0, 4.0};    /* y = -1 + 4i */
Complex z;

z = add(x, y);              /* z = 2 + 9i */
```

The declaration of the function `add()` would occur after the declaration of the structure `Complex` in the header file `Complex.h`. Similarly, its definition should occur in the source file `Complex.c`. This code is relatively natural, as the programmer is using the concept of a complex number, without referring to the underlying implementation details.

16.4 Pointers to Structures

Just as you can declare pointers to integers, floating point numbers and characters, you can also declare pointers to structures:

```
Student *pstd;
University *puni;
Complex *pcomp;
```

Here the `*` (star) notation tells the compiler that a pointer is being declared rather than a variable and the `p` at the start of the name is just to remind you that this is a pointer.

Memory can be allocated to a structure using `malloc()` in the following manner:

```
pstd = (Student *) malloc(sizeof(Student));
pcomp = (Complex *) malloc(20*sizeof(Complex));
```

which reserves memory for one `Student` and an array of 20 variables of type `Complex`. Calls to `free()` deallocate this memory:

```
free(pstd);
free(pcomp);
```

Note that you can have pointers to structures as members of other structure. For instance, it would have been better to declare `std` as a pointer to a `Student`, rather than as an array in the definition of `University`. This would then allow the correct amount of memory to be allocated at run-time rather than fixing its size at compile-time. To declare a structure as a member of another structure, the former needs to be declared before the latter as memory has to be allocated when a variable of that type is declared. However, you can declare a pointer to a structure that hasn't been defined and this is because *all* pointers use up the same amount of memory. Structures can even contain pointers to themselves, and this is a common technique for constructing *linked lists* in C.

16.4.1 Accessing Members

If you have declared a pointer to a structure and allocated memory to it with a call to `malloc()`, its members can be accessed using the arrow `->` notation (a hyphen followed by a greater than character). This is illustrated in the following code segment:

```
Complex *pcomp;
pcomp = (Complex *) malloc(sizeof(Complex));
pcomp->real = 3.0;
pcomp->im = -5.0;
```

Instead of using a dot to reference the member of a structure, an arrow is used instead, and this is because the two following expressions are equivalent:

```
pcomp->real
(*pcomp).real
```

The dereferencing * operator means that the pointer can now be treated as a true variable, and the dot operator can be used to access the member. However, because the dot operator has a higher precedence than the star operator, the first part of the expression needs to be placed in parentheses. This notation is obviously cumbersome, so the arrow notation is used instead.

Although C provides a mechanism to copy and assign structures, pointers are generally used to transfer information instead. Note that the only way to pass an array to a function is via pointers. The reason for this is that structures and arrays generally contain a lot of data and to copy all of the variables each time a function is called is *slow*. Pointers are used to pass the location of the array/structure in memory and this is obviously much quicker, although sometimes the notation can be confusing!

16.4.2 Initialising and Destroying Structures

Just like functions are used to hide the complexity of a procedure, structures are used to hide the complexity inherent in the data. For instance, you may write a function that implements a Newton-Raphson calculation and the user of such a function does not need to understand how Newton-Raphson works. Similarly, you may define a structure of type `Student` with a set of well-defined interface functions and the user of this data type would not need to understand how this has been implemented. In practice, both of these techniques are used to design large programs as a programmer can work on one small subtask at one time, and it unnecessary to understand the whole problem. In fact, problem statements are often so poorly defined that the programmer must enter an interactive design cycle, as discussed in the next section. So the ability to encapsulate both procedures and data into smaller, well-defined parcels is extremely good programming style.

Once a pointer to a structure is defined, it is usual to declare and define *initialisation* and *destroying* functions that are called just after it is declared and before it goes out of scope. As these are *interface* functions (analogous to `fopen()` and `fclose()`), it is common to declare them in the header file just after the structure is defined.

For the `Student` structure, these functions would be declared as:

```
Student *Student_init(char *name, int age, int mark1, int mark2,
                      int lectures_missed, double bank_balance);
void Student_destroy(Student *pstd);
```

and defined something like:

```
/* Allocate memory and initialise a Student variable
 * using the arguments supplied. Returns a pointer
 * to allocated memory.
 */
Student *Student_init(char *name, int age, int mark1, int mark2,
                      int lectures_missed, double bank_balance)
{
  Student *pstd;
                          /* reserve space for a type Student */
  pstd = (Student *) malloc(sizeof(Student));
  assert(pstd != NULL);
```

```
                        /* reserve space for *name */
  pstd->name = (char *) malloc((strlen(name)+1)*sizeof(char));
  assert(pstd->name != NULL);
  strcpy(pstd->name, name);     /* copy string */
                        /* copy remaining variables */
  pstd->age  = age;
  pstd->bank_balance  = bank_balance;
  pstd->assn_marks[0] = mark1;
  pstd->assn_marks[1] = mark2;
  pstd->no_lectures_missed  = lectures_missed;

  return pstd;      /* return pointer to reserved memory */
}


/* Free the memory allocated to a pointer of type
 * Student by first releasing name and then pstd.
 */
void Student_destroy(Student *pstd)
{
  free(pstd->name);
  free(pstd);

  return;
}
```

Even when a variable is declared it is common to define initialisation and freeing functions that allocate and free any internal dynamic memory.

Functions and structures are generally grouped together in large programs and a set of functions are written to manipulate (initialise, delete, modify) each structure, where the structure and its functions are generally grouped together in a separate source file, with the header file specifying this interface. This structured paradigm then leads naturally onto *Object-Oriented programming*, and C++, which is the most widely used OO language.

16.4.3 Structures and Functions

The final reason for using structures is that their use can hide the amount of data being passed between functions. Instead of having to pass all the members individually, only a single variable (or a pointer to that variable) needs to be passed. This reduces the chance of making typing errors or putting the function's arguments in the wrong order as this is hidden in the structure's definition. Also, if you add a new member to a structure, the function's interface will *not* change, as it will be implicitly passed across along with the rest of the structure. This feature means that C supports an *incremental programming* strategy where the motto is:

> *Don't break working code.*

Code should be designed so that the minimum amount of changes are necessary when a new feature is added.

There are two ways to declare an instance of a structure: either as a variable or as a pointer, and many people are confused over which one is best. The rule of thumb is that if you're structure is large or contains pointers or arrays, declare a pointer to that structure and define your own initialisation functions, otherwise just declare a normal variable and use C's inbuilt copying and assignment rules.

16.5 Example: A Vector Library

To illustrate some of the ideas discussed in this section, a vector object will be created that can be used with other source files. It stores and manipulates a vector of type `double`. The `Vector` structure is similar to a 1 dimensional array to type `double`, except that it remembers its own length, and also a set of initialisation, destroying and manipulation functions are also provided.

16.5.1 Example: `Vector.h`

```
/* Header file for a double Vector library
 */

#ifndef VECTOR_H
#define VECTOR_H

#include <stdio.h>          /* necessary for FILE* */
                            /* declare Vector structure */
typedef struct Vector {
  int length;     /* store vector's length */
  double *parr;             /* pointer to a 1-d array */
} Vector;

Vector Vec_init_filename(char *name);     /* create from file name */
void   Vec_destroy(Vector *pvec);         /* destroy memory */
void   Vec_print(Vector vec);             /* print to screen */

Vector Vec_add(Vector v1, Vector v2);
Vector Vec_subtract(Vector v1, Vector v2);
double in_prod(Vector v1, Vector v2);

#endif
```

16.5.2 Examination of `Vector.h`

The most important issue to decide when the header file was created was whether the structure or a pointer to the structure was going to be used to initialise and access information. The structure is small so the copying overhead associated with passing the structure will not be too significant, but the structure contains a pointer to a 1-D array of type `double`, so it may be better to force the user to pass across pointers. In the end, I decided to design the `Vector` library based on passing structures because it is more natural to code, but if running speed was critical, a set of alternative functions based on pointers could be designed. This is left as an exercise.

Two interface functions are provided that ensure that variable of type `Vector` is initialised and destroyed correctly, and three basic operations are coded: adding and subtracting two vectors and taking two vectors' inner product. In addition, a small function has been written which prints the elements contained in the vector to the screen. This is very useful during the early stages of code development, as it can be used to check that all the other functions are working correctly.

When you are designing a library of routines such as this, it is essential that you give your structure and its functions *meaningful* names, otherwise they may clash with functions defined in other libraries.

16.5.3 Example: `Vector.c`

```
#include "Vector.h"           /* include interface definitions */
#include <assert.h>           /* assert() function used in source code */

Vector Vec_init_memory(int len);     /* function internal to this file */


/* Initialise a vector from a file with supplied filename
 */
Vector Vec_init_filename(char *name)
{
  int len;
  FILE *read;
  Vector vec = {0, NULL};            /* assign default values */

  read = fopen(name, "r");           /* open file */
  assert(read != NULL);

  fscanf(read, "%d", &len);   /* read in vector's size */

  vec = Vec_init_memory(len);
                                     /* read init values from file */
  for (i = 0; i < vec.length; i++)
    fscanf(read, "%lf", &vec.parr[i]);

  fclose(read);
                                    /* return copy of vec */
  return vec;
}

/* Set up memory for a vector of length len
 */
Vector Vec_init_memory(int len)
{
  int i;
  Vector vec = {0, NULL}

  assert(len >= 0);                  /* set length */
  vec.length = len;
                                     /* allocate memory */
  vec.parr = (double *) malloc(vec.length*sizeof(double));
  assert(vec.parr != NULL);
                                     /* initialise memory */
  for (i = 0; i < vec.length; i++)
```

```
    vec.parr[i] = 0.0;

  return vec;
}


/* Free the memory associated with vec
 */
void Vec_destroy(Vector *pvec)
{
  if (pvec->length == 0 && pvec->parr == NULL)
    return;
  else {
    free(pvec->parr);
    pvec->length = 0;       /* once deleted, set default values */
    pvec->parr = NULL;
  }

  return;
}


/* Print the length of the vector and the
 * individual elements to the screen.
 */
void Vec_print(Vector vec)
{
  int i;

  printf("length of vector = %d\n", vec.length);
  for (i = 0; i < vec.length; i++)
    printf("v[%d] = %f\n", i, vec.parr[i]);

  return;
}


/* Add two vectors together
 */
Vector add_Vec(Vector v1, Vector v2)
{
  int i;
  Vector vec;

  assert(v1.length == v2.length);
                                        /* initialise memory for vec */
  vec = Vec_init_memory(v1.length);

  for (i = 0; i < vec.length; i++)
    vec.parr[i] = v1.parr[i]+v2.parr[i];
```

```
  return vec;
}


/* Subtract v2 from v1 (where v1 & v2 are vectors)
 */
Vector subtract_Vec(Vector v1, Vector v2)
{
  int i;
  Vector vec;

  assert(v1.length == v2.length);

  vec = Vec_init_memory(v1.length);

  for (i = 0; i < vec.length; i++)
    vec.parr[i] = v1.parr[i]-v2.parr[i];

  return vec;
}


/* Calculate the inner product of two vectors
 */
double in_prod(Vector v1, Vector v2)
{
  int i;
  double result = 0.0;
                                   /* check vectors are same length */
  assert(v1.length == v2.length);

  for (i = 0; i < v1.length; i++)
    result += v1.parr[i]*v2.parr[i];

  return result;
}
```

16.5.4 Examination of `Vector.c`

A substantial amount of work is involved with providing a set of flexible interface functions, and this should be obvious from this example. This construction provides a fairly natural interface for the library user, except that after each call to `Vec_add()` or `Vec_subtract()`, the memory must be explicitly deallocated with a call to `Vec_destroy()` before the same variable can be reused. This will now be illustrated.

Note that the function `Vec_init_memory()` has been declared at the top of the source file, rather than in the header file. This means that it can be called from within the source file `Vector.c`, but other source files are not able to access it when the header file `Vector.h` is included. (Strictly speaking, we should preceed its declaration by the keyword `static`, because other files can still access it using the `extern` keyword.)

16.5.5 Example: main()

```
#include <stdlib.h>
#include "Vector.h"                     /* include interface definitions */

int main()
{
  Vector vec1 = {0, NULL};              /* default values */
  Vector vec2 = {0, NULL};
  Vector vec3 = {0, NULL};
                                        /* initialise vec1 and vec2 */
  vec1 = Vec_init_filename("vector1.txt");
  vec2 = Vec_init_filename("vector2.txt");
                                        /* add two vectors */
  vec3 = Vec_add(vec1, vec2);

  Vec_destroy(&vec3);                   /* reset memory for v3 */
  vec3 = Vec_subtract(vec1, vec2);       /* v1-v2 */

  Vec_destroy(&vec1);                   /* clear memory at */
  Vec_destroy(&vec2);                   /* end of program */
  Vec_destroy(&vec3);

  return EXIT_SUCCESS;
}
```

16.6 Summary

- The `struct` keyword allows programmers to create user-defined data types which can hide unnecessary data complexity from the user of a software library.
- Structures are generally declared *globally* (in a header file, for instance) which allows variables of that type to be instantiated in other files if necessary.
- For a variable, the member of a structure is accessed using the `.` operator, whereas for a pointer to a structure, the member is accessed using the `->` operator.
- A structure can be copied can assigned to another variable, and this can be extremely useful as a function may return a structure which contains several variables. Similarly, passing a structure (or a pointer to a structure) to a function can vastly simplify the argument list.
- Memory for structures can be allocated and deallocated using `malloc()` and `free()`, as the `sizeof` operator can also be used for user-defined structures.
- A software library usually consists of one or more structures (data type) and a set of functions which act as an *interface* (creating, modifying, destroying) for these structures. The structures and functions should be declared in the header file and defined in the source file.

17 Software Engineering with C

So far in this course, you have learned about the *semantics* and the *syntax* of the C programming language. It turns out that only a few well defined concepts are needed to provide a foundation for most programming languages, and this is reflected by the surprisingly(?) small number of keywords in the C language. In fact, a lot of a programming language's baggage (structures, functions etc.) has evolved to make it *easier* for programmers to manage the data and functional complexity explosion that occurs in larger programs. You have learned how to use these basic ideas, but as was mentioned in section 1, this course has not concentrated on the ideas behind *software engineering*. All software engineers are programmers but not vice versa, as software engineers use well-defined practices to construct software which makes it easier to design, develop and maintain. In this final section, we shall have a look at the basic ideas behind software engineering and investigate how C programs can be designed to adhere to these conventions.

17.1 Software Life Cycle

One of the problems with describing the software engineering discipline is that few students have ever worked on developing large software products, so they find it difficult to appreciate the problems that are encountered. In particular:

- All the exercises and assignments are well-defined in that there is always an answer, although frequently there are several!

- You do (should?) not cooperate on the software development, and the programs do not take more than a couple of days to write.

- You don't have to maintain the programs, or modify them weeks after they have been written.

However, software engineers are frequently faced with ill-defined problems, they must work in teams to ensure that deadlines are met and they must maintain the programs and respond to users' queries and demands for new releases. These facts must be recognised when developing large pieces of software and this is emphasied in the *waterfall* and *spiral* models of the software design process.

17.1.1 Waterfall Model

The *waterfall model* of the software *life cycle* is shown in Figure 21. This diagram has 5 main blocks:

needs analysis and specification where the set of requirements that the software should fulfill are obtained from the customer,

design where the major datatypes and operators (ie. data abstraction) are identified along with appropriate interfaces,

implementation is the process you're familiar with as it refers to implementing the design in the most appropriate language, although more often or not its the programmer's favourite language,

testing where the program's functionality is tested for correctness and generality (remember the Hubble telescope?) and

maintenance and upgrades where the programs must be maintained, ported to new systems and its functionality increased.

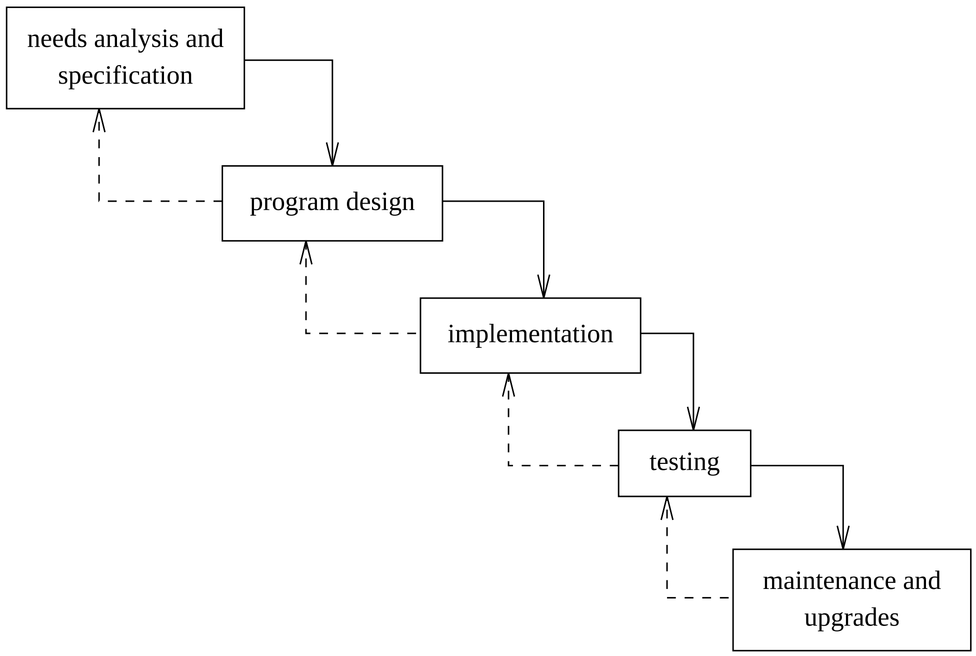

Figure 21: The waterfall model of the software life cycle which incorporates an amount of reengineering (dashed lines).

and as you can see from the diagram, there is a constant flow from one block to the next as well as a continuous assessment to see how well the previous subtask has been performed, and if necessary go back to modify that module.

17.1.2 Spiral Model

The spiral model for software engineering has evolved to encompass the best features of the classic waterfall model, while at the same time adding an element known as *risk analysis*. The spiral model is more appropriate for large, industrial software projects and has four main blocks/quadrants, as illustrated in Figure 22:

planning where the objectives, alternatives and constraints are determined,

risk analysis where the potential problems are identified and alternative solutions are analysed,

engineering where the next release is developed, and

customer evaluation where the software is assessed for features and usability.

Each release or version of the software requires going through new planning, risk analysis, engineering and customer evaluation phases and this is illustrated in the model by the spiral evolution outwards from the centre. For each new release of a software product, a risk analysis audit should be performed to decide whether the new objectives can be completed within budget (time and costs), and decisions have to be made about whether to proceed. The level of planning and customer evaluation is missing from the waterfall model which is mainly concerned with small software programs.

The spiral model also illustrated the *evolutionary* development of software where a solution may be initially proposed which is very basic (first time round the loop) and then later releases add new features and possibly a more elaborate GUI.

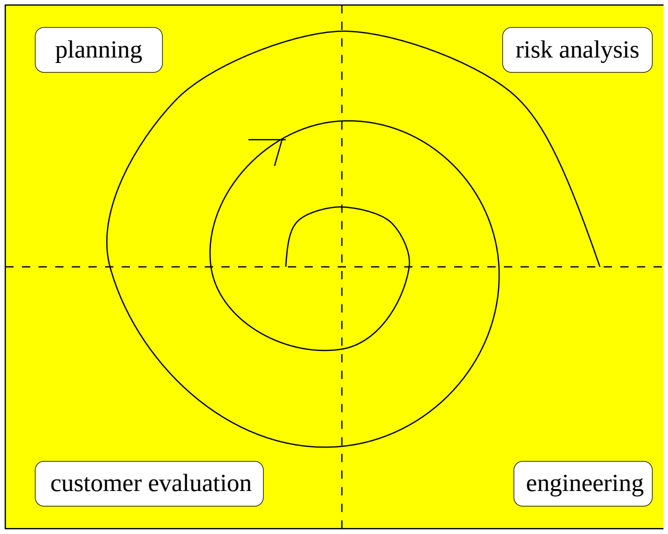

Figure 22: The spiral model of the software life cycle which includes an evolutionary element.

17.2 Modularity

As shown in the second module of the software life cycle, *design* is an important process in software engineering. The design process amounts to producing an appropriate software abstraction of real-world objects and procedures that will be used in implementation phase. This abstraction process is fairly generic in the sense that the work done should be insensitive to the final implementation language, although some languages support data and functional abstraction concepts better than others.

17.2.1 Functional Paradigm

Data and functional abstractions are different, but related, ideas. During the sixties and seventies, functional languages were seen as one way of overcoming the *software crisis*, where large software projects were failing because the programming tools were inappropriate for the job. Functional languages, such as LISP, emphasise the process of identifying blocks, and repeated pieces, of code (such as reading from a file or calculating a square root of a number) and to construct functions (sometimes called procedures or subroutines) that encapsulate the functionality within a single definition. This allowed programs to be written that were more:

maintainable because the definition of this routine only occurs once in the program

reliable as no typing errors occur in other definitions of the same routine.

predictable because a function generally represents a distinct data transformation that is easily understood by the designer.

Functional languages have taken this idea to an extreme, as their syntax encourages a totally modular approach to program flow. It reuses previously written code (functions) to build more complex programs and generally there is no `main()` function (or equivalent) to denote the point where the program execution begins. However, by focussing totally on the flow of data, rather than the internal organisation of data, functions can become "brittle" over time as their public interface changes.

17.2.2 Object Paradigm

In the late seventies/early eighties it became apparent that similar abstraction mechanisms must be developed to support data as well as functions, and this is achieved in C with user-defined structures. More recently, *object oriented* languages, such as Smalltalk and C++, provide a greater support for data abstraction and indeed, integrate functions and data together into a single *class* or *object*.

Data abstraction relies on identifying relevant, real-world concepts that can be used to partition the overall problem into a set of loosely related sub-problems. This can be illustrated by considering the `University` and `Student` structures described in section 16. Suppose the problem was to develop a database for the records of a university, we would identify the parts it was composed of:

> *students, lecturers, support staff, buildings, faculties, departments, offices, libraries, bars, sports facilitates, etc.*

Once these elements were identified, we could start working out their interrelationships. For instance, we could say that a university has many departments, and declare a vector of `Department`s inside the `Univeristy` structure. However, it would be more correct to say that a university is composed of many `Faculty` structures and each `Faculty` is made of several `Department`s. Selecting appropriate objects in the program design phase ensures that function interfaces should not change, even if the internal composition of the structure and the function does. These relationships can be represented pictorially using an *entity-relationship* notation. The two most important types of relationships are the *is a* and the *has a*, and examples of these types of relationships are: *A library is a building*

A university has many students

and the difference of meaning between these two examples, emphasises how modular data structures may be constructed.

By saying that a `Library` is a `Building`, we can abstract concepts that are relevant to a building such as an address, the number of rooms it contains etc., and place this data in the `Building` structure. To define a library, we can reuse this definition by incorporating a `Building` variable as a data member inside the `Library` structure, and any functions we write to set an address etc., for a `Building` can also be used to set an address for a library. This type of data structuring promotes function reuse.

Identifying *has a* relationships provides a means for developing appropriate functional interfaces. If we wanted to add another student to a university (or conversely expel one), the easiest way to achieve this is to pass across the array of `Student` structures to an appropriate function that either tags one on the end or deletes a member. Note that you're not passing across the whole `University` structure, only one of its members. If instead, the `University` structure had been declared with no `Student` members, only the appropriate arrays for the individual bank balances, ages, names etc., then either the `University` structure would have to be passed to the function or each of the individual arrays; for either case this is undesirable. For the latter option, adding another feature to describe a student would cause the function's interface to change and this would signify a badly designed program. For the former method, too much information is being passed to the function and the programmer cannot guarantee that any of this extra data will not be changed as a side effect of the function call.

Describing the program's data structures using *is a* and *has a* relationships is probably the main tool for identifying real-world concepts that are useful.

17.2.3 Software Libraries

Software libraries consist of related functions and data structures that a programmer may find useful when writing general programs. The philosophy behind C and C++ is that the number of keywords (such as `for`, `if`, etc.) in the language should be kept to a minimum and support for other types of functionality (file input/output, complex numbers etc.) should come from software libraries. This strategy has proved extremely successful as it has ensured that C compilers are available on a wide range of platforms and they are generally *small*. In addition, a large third-party business has grown to support various libraries (in addition to the ones supplied by the compiler), and true to our free market economy ideals, the suppliers provide what the programmers need.

In the C language, header files are used to declare the public interface elements of the corresponding library, and the source file holds the actual definition. Therefore, performing the program design phase of software engineering amounts to identifying appropriate structures and functions which are then declared in (possibly) several header files. During this design phase, the corresponding source files do not have to be written (that is done in the implementation phase) and by concentrating on the high-level design, rather than the low-level code writing, this should result in programs that are easier to maintain and extend.

By splitting your program into several files or libraries, you obtain the following advantages:

- The programmer can concentrate on implementing a specific task, without worrying about how the library will be used.
- As part of a larger team, each programmer can work on their individual library and then each section is *linked* together to form the complete program.
- Compiling each source file individually means that small changes can be made in one library and there is no need to recompile the whole program. Only the appropriate module needs to be recompiled.
- Unix provides a `maketool` facility to manage the compilation process and equivalent PC tools also exist.
- Supports the top-down/bottom-up design philosophy where the interface requirements (header file) are determined top-down and the implementation (source file) is written bottom-up.
- Libraries can be *shared* between languages so any time spent on designing a good C library will not be wasted as it can be linked with C++ and FORTRAN object code.

An example is given in section 16.5 where a vector library is constructed. Once the desired functionality is encoded in the header file, the `main()` function can be written based on these interfaces under the assumption that the appropriate object file will be linked in later on.

This software writing process embodies an incremental design philosophy, where a set of consistent interfaces are produced before the low-level details are implemented. The consistent interfaces are based on high-level concepts (structures) such as `Vector`, `University`, `Student` etc. and so even if the low-level detail changes, the functional interface won't. This allows the designer to prototype systems using almost pseudo-code language, without specifying the underlying detail.

Commercial software houses produce products that are upgraded every year or so. This allows them to obtain revenues from an initially basic package and then to add on new features at each upgrade. It is "impossible" to design the perfect word processor or spreadsheet, and software designers acknowledge this fact. Hence, they adopt processes that are sufficiently general to allow the software to evolve in functionality which ensures that a minimum amount of rewriting is involved when adding new features.

17.3 Software Constructs

In this section, we shall briefly review some design aids that can help programmers to approach large, complex tasks.

17.3.1 Top-Down and Bottom-Up Design

In many engineering tasks, designers often talk about top-down or bottom-up solutions to their problem. In a top-down design, an engineer would provide a short summary of the problem and solution, and identify each significant subtask. The next step is to try and solve each subtask individually, and these in turn may be broken down into sub-subtasks. Overall, a hierarchical structure is constructed such that each module solves a small part of the original program. This is probably the most obvious paradigm for encapsulating details into well-defined blocks, but more recently, recognition has been given to the merits associated with bottom-up design.

Rather than focussing the problem's structure, bottom-up designers search for generic tools, or libraries, that may prove useful in the solution to more complex problems. Often this type of design was performed implicitly, but recognising its importance emphasises the datatypes that may be useful in the program's design.

In practice, both philosophies are used to design useful programs. For instance, in trying to invert a matrix, the top-down methodology would probably focus on how to code a specific method for inverting a matrix, whereas the bottom-up designer would first design a set of `Vector` and `Matrix` classes, and then implement the necessary functionality to actually perform the matrix inversion. It is easy to see that both parts are necessary to perform the desired task, but by focussing on building `Vector` and `Matrix` libraries, the code can be reused in different programmes.

17.3.2 Nouns and Verbs

Whether a top-down or bottom-up design is used and whether the designer adopts a functional or object paradigm, they still have to decide on what functions and what (abstract) data types should be used to solve the problem. A simple but effective idea which has proved popular over the years is to first write down the problem and its solution using natural language terms and then identify the *nouns* with abstract data types (structures or objects) and the *verbs* with functions. Sentences such as:

> *Read in a set of students' records from a database.*

would imply that you constructed a structure for each *student's records* (and then declare an array of these structures), and wrote a function which *read* them from some *database*. Other functions would need to be written to which initialised and destroyed the structures, but by associating functions with transformations of the noun's state, reasonably detailed software design diagrams can be constructed directly from a written problem statement.

As well as deciding which functions are necessary in a program, the relationships between the nouns can be described using the *is a* and *has a* relationships. This ensures that most of the abstract data types that will be created in a program will have a direct interpretation in terms of real-world concepts and make the program easier to design and develop but most importantly to *maintain*.

17.3.3 Dataflow diagrams

Dataflow diagrams are used in the analysis and design of software to pictorially represent both the program's flow and the state of its datatypes. The symbols in the diagram have specific meanings:

Arrows represent data paths.

Circles represent data manipulation functions.

Rectangles represent data sources (such as humans).

Heavy straight lines represent data storage (files, database structures etc.)

Dataflow diagrams are extremely useful as it forces the program designer to think about how to split up his problem in a top-down fashion. Of course, you also need to extend the routines in a "bottom-up" fashion to make them reasonably generic.

17.4 Further Reading

Software design will always be a hotly debated topic as new ideas get tested on large projects and the successful ones are refined and form the basis for the semantics and the syntax of new programming languages. Object-oriented programming techniques, which is currently the flavour of the month, grew out of the AI community and languages such as SIMULA. These paradigms are now widely used in the conventional software engineering community, but limitations are being found in the object design process and modifications are being proposed. There are many journals devoted to these topics, and the library subscribes to several. For an extensive discussion about software engineering, see:

Pressman, R.S. 1994. *Software Engineering: A Practitioner's Approach*, 3rd Edition, McGraw-Hill, London.

17.5 Summary

- Software engineering isn't an exact science and is best learnt by experience. However, there are some guidelines which can be applied.
- The software life cycle and spiral models are only a guide to how software evolves. Each project is different.
- Modular (data and functional) software design is very important as it allows several programmers to work together on the same project, each specialising in a certain area. In addition, programs become more *maintainable* because the effects of changes to the code are localised.
- *is a* and *has a* relationships can be used to model the interdependency between structures, where a structure corresponds to a *noun* in the problem solution and a function corresponds to a *verb*.
- In practice, top-down and bottom-up design procedures are combined to produce code which is appropriate for the current project but which may also be extended by later projects. Note that a clear understanding of the job of the code module is necessary, if it is going to be extended later on.

A Keywords

This appendix contains a full list of keywords used in the ANSI C language. The first set of keywords are for declaring variables, the second set are for declaring new data types and the third are used to control the flow of the program. In addition, the `sizeof` operator (keyword) can be used to find the size (in bytes) of a variable or datatype.

Variable Declaration Keywords

```
auto      char      const     double    extern
float     int       long      register  short
signed    static    unsigned  void      volatile
```

Data Type Declaration Keywords

```
enum      struct    typedef   union
```

Program Flow Control Keywords

```
break     case      continue  default   do
else      for       goto      if        return
switch    while
```

B The ANSI C Standard Library

This appendix contains a listing and short explanation of some of the functions contained in the ANSI C standard library. If you are using these extensively, it is recommended that you purchase a reference book, such as *The C Programming Language* (2^{nd} edition) by Kernighan and Ritchie. This appendix is based on the ANSI C Standard Libraries document by Ross Richardson at:
`http://www.cs.utas.edu.au/Documentation/C/CStdLib.html`

To use the functions contained in these libraries, it is necessary to include the releavant header file at the start of your program, ie:

```
#include <library_name>
```

Header files, by convention, end with a `.h` suffix, and on the Solaris system, they are contained in the directory:

```
/usr/include
```

and an individual file can be read in a texteditor, for instance. On the Solaris system, it is also possible to find out about some of the functions and libraries by using the `man`ual pages and typing:

```
man name
```

where name is the name of the function or header file (without the `.h` extension).

B.1 `<assert.h>`

`void assert(int expression);`
: *Macro* used to add diagnostics. If `expression` is false, message printed on `stderr` and `abort` called to terminate execution. Source file and line number in message come from preprocessor macros `__FILE__` and `__LINE__`. If `NDEBUG` is defined where `<assert.h>` is included, `assert` macro is ignored.

B.2 `<ctype.h>`

These functions allow the user to test characters and determine their type (space, number, upper-case, etc.). The functions return a non-zero value (true) when the condition is satisfied, and zero otherwise.

`int isalnum(int c);`
: true when `isalpha(c)` or `isdigit(c)` is true

`int isalpha(int c);`
: true when `isupper(c)` or `islower(c)` is true

`int iscntrl(int c);`
: true when `c` is a control character

`int isdigit(int c);`
: true when `c` is a decimal digit

`int isgraph(int c);`
  true when `c` is a printing character other than space

`int islower(int c);`
  true when `c` is a lower-case letter

`int isprint(int c);`
  true when `c` is a printing character (including space)

`int ispunct(int c);`
  true when `c` is a printing character other than space, letter, digit

`int isspace(int c);`
  true when `c` is space, formfeed, newline, carriage return, tab, vertical tab

`int isupper(int c);`
  true when `c` is upper-case letter

`int isxdigit(int c);`
  true when `c` is hexadecimal digit

Note In ASCII (7-bit), printing characters are `0x20 (' ')` to `0x7E ('~')`; control characters are `0x00 (NUL)` to `0x1F (US)` and `0x7F (DEL)`

The following two functions swap a letter's case:

`int tolower(int c);`
  return lower-case equivalent

`int toupper(int c);`
  return upper-case equivalent

B.3 `<float.h>`

`FLT_RADIX`

`FLT_ROUNDS`

`FLT_DIG`

`FLT_EPSILON`
  smallest number `x` such that `1.0 != 1.0 + x`

`FLT_MANT_DIG`

`FLT_MAX`
  maximum floating-point number

`FLT_MAX_EXP`

`FLT_MIN`
  minimum normalised floating-point number

FLT_MIN_EXP

DBL_DIG

DBL_EPSILON

DBL_MANT_DIG

DBL_MAX
: maximum `double` floating-point number

DBL_MAX_EXP

DBL_MIN
: minimum normalised `double` floating-point number

DBL_MIN_EXP

B.4 `<limits.h>`

CHAR_BIT
: number of bits in a `char`

CHAR_MAX
: maximum value of `char`

CHAR_MIN
: minimum value of `char`

INT_MAX
: maximum value of `int`

INT_MIN
: minimum value of `int`

LONG_MAX
: maximum value of `long`

LONG_MIN
: minimum value of `long`

SCHAR_MAX
: maximum value of `signed char`

SCHAR_MIN
: minimum value of `signed char`

SHRT_MAX
: maximum value of `short`

SHRT_MIN
: minimum value of `short`

`UCHAR_MAX`
maximum value of `unsigned char`

`UCHAR_MIN`
minimum value of `unsigned char`

`UINT_MAX`
maximum value of `unsigned int`

`ULONG_MAX`
maximum value of `unsigned long`

`USHRT_MAX`
maximum value of `unsigned short`

B.5 `<math.h>`

A lot of these functions are fairly self-explanatory, but remember that when this maths library is included on the Unix systems, you must add the `-lm` flag when you compile the program, otherwise you'll get an undefined symbol error. The arguments and return values for the trignometic functions are expressed in *radians*.

`double sin(double x);`

`double cos(double x);`

`double tan(double x);`

`double asin(double x);`
$\sin^{-1}(x) \in [-\pi/2, \pi/2]$ when $x \in [-1, 1]$

`double acos(double x);`
$\cos^{-1}(x) \in [0, \pi]$ when $x \in [-1, 1]$

`double atan(double x);`
$\tan^{-1}(x) \in [-\pi/2, \pi/2]$

`double atan2(double y, double x);`
$\tan^{-1}(y/x) \in [-\pi, \pi]$

`double sinh(double x);`

`double cosh(double x);`

`double tanh(double x);`

`double exp(double x);`

`double log(double x);`
natural logarithm $\ln(x)$

`double log10(double x);`

`double pow(double x, double y);`
: `x` raised to power `y`

`double sqrt(double x);`
: square root

`double ceil(double x);`
: smallest integer not less than `x`

`double floor(double x);`
: largest integer not greater than `x`

`double fabs(double x);`
: absolute value

`double ldexp(double x, int n);`

`double frexp(double x, int* exp);`

`double modf(double x, double* ip);`

`double fmod(double x, double y);`

B.6 <setjmp.h>

`int setjmp(jmp_buf env);`
: Save state information in `env`. Zero returned from direct call; non-zero from subsequent call of `longjmp`.

`void longjmp(jmp_buf env, int val);`
: Restore state saved by most recent call to `setjmp` using information saved in `env`. Execution resumes as if `setjmp` just executed and returned non-zero value `val`.

B.7 <signal.h>

Handling exceptional conditions.

`SIGABRT`
: abnormal termination

`SIGFPE`
: arithmetic error

`SIGILL`
: illegal function image

`SIGINT`
: interactive attention

`SIGSEGV`
: illegal storage access

`SIGTERM`
termination request sent to program

`void (*signal(int sig, void (*handler)(int)))(int);`
Install handler for subsequent signal `sig`. If `handler` is `SIG_DFL`, implementation-defined default behaviour is used; if `handler` is `SIG_IGN`, signal is ignored; otherwise function pointed to by `handler` is called with argument `sig`. `signal` returns the previous handler or `SIG_ERR` on error. When signal `sig` subsequently occurs, the signal is restored to its default behaviour and the handler is called. If the handler returns, execution resumes where signal occurred. Initial state of signals is implementation-defined.

`int raise(int sig);`
Send signal `sig` to the program. Non-zero returned if unsuccessful.

B.8 `<stdarg.h>`

Facilities for stepping through a list of function arguments of unknown number and type.

`void va_start(va_list ap, lastarg);`
Initialisation macro to be called once before any unnamed argument is accessed. `ap` must be declared as a local variable, and `lastarg` is the last named parameter of the function.

`type va_arg(va_list ap, type);`
Produce a value of the type (`type`) and value of the next unnamed argument. Modifies `ap`.

`void va_end(va_list ap);`
Must be called once after arguments processed and before function exit.

B.9 `<stdio.h>`

`FILE` Type which records information necessary to control a stream.

`stdin`
Standard input stream. Automatically opened when a program begins execution.

`stdout`
Standard output stream. Automatically opened when a program begins execution.

`stderr`
Standard error stream. Automatically opened when a program begins execution.

`FILENAME_MAX`
Maximum permissible length of a file name

`FOPEN_MAX`
Maximum number of files which may be open simultaneously.

`TMP_MAX`
Maximum number of temporary files during program execution.

`FILE* fopen(const char* filename, const char* mode);`
Opens file `filename` and returns a stream, or `NULL` on failure. `mode` may be (combinations of):

`"r"`
: text reading

`"w"`
: text writing; discard previous content

`"a"`
: text append; writing at end

`"r+"`
: text update

`"w+"`
: text update; discard previous content

`"a+"`
: text append; writing at end

`FILE* freopen(const char* filename, const char* mode, FILE* stream);`
: Opens file `filename` with the specified mode and associates with it the specified stream. Returns `stream` or `NULL` on error. Usually used to change files associated with `stdin`, `stdout`, `stderr`.

`int fflush(FILE* stream);`
: Flushes stream `stream`. Effect undefined for input stream. Returns `EOF` for write error, zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE* stream);`
: Closes stream `stream` (after flushing, if output stream). Returns `EOF` on error, zero otherwise.

`int remove(const char* filename);`
: Removes file `filename`. Returns non-zero on failure.

`int rename(const char* oldname, const char* newname);`
: Changes name of file `oldname` to `newname`. Returns non-zero on failure.

`FILE* tmpfile();`
: Creates temporary file (mode `"wb+"`) which will be removed when closed or on normal program termination. Returns stream or `NULL` on failure.

`char* tmpname(char s[L_tmpnam]);`
: Assigns to `s` and returns unique name for temporary file.

`int setvbuf(FILE* stream, char* buf, int mode, size_t size);`
: Controls buffering for stream `stream`.

`void setbuf(FILE* stream, char* buf);`
: Controls buffering for stream `stream`.

`int fprintf(FILE* stream, const char* format, ...);`
: Converts (with format `format`) and writes output to stream `stream`. Number of characters written [negative on error] is returned. Between

- Flags:

  `-`
  : left adjust

  `+`
  : always sign

space space if no sign

`0`
: zero pad

`#`
: Alternate form: for conversion character `o`, first digit will be zero, for [`xX`], prefix `0x` or `0X` to non-zero, for [`eEfgG`], always decimal point, for [`gG`] trailing zeros not removed.

- Width:
- Period:
- Precision: for conversion character `s`, maximum characters to be printed from the string, for [`eEf`], digits after decimal point, for [`gG`], significant digits, for an integer, minimum number of digits to be printed.
- Length modifier:

`h`
: `short` or `unsigned short`

`l`
: `long` or `unsigned long`

`L`
: `long double`

Conversions :

`d,i`
: `int`; signed decimal notation

`o`
: `int`; unsigned octal notation

`x,X`
: `int`; unsigned hexadecimal notation

`u`
: `int`; unsigned decimal notation

`c`
: `int`; single character

`s`
: `char*`;

`f`
: `double`; [-]mmm.ddd

`e,E`
: `double`; [-]m.dddddde(+—-)xx

`g,G`
: `double`

`p`
: `void*`; print as pointer

`n`
: `int*`; number of chars written into arg

`%`
: print `%`

`int printf(const char* format, ...);`
`printf(f, ...)` is equivalent to `fprintf(stdout, f, ...)`

`int sprintf(char* s, const char* format, ...);`
Like `fprintf()`, but output written into string `s`, which must be large enough to hold the output, rather than to a stream. Output is `NULL`-terminated. Return length does not include the `NULL`.

`int vfprintf(FILE* stream, const char* format, va_list arg);`
Equivalent to `fprintf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int vprintf(const char* format, va_list arg);`
Equivalent to `printf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int vsprintf(char* s, const char* format, va_list arg);`
Equivalent to `sprintf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int fscanf(FILE* stream, const char* format, ...);`
Performs formatted input conversion, reading from stream `stream` according to format `format`. The function returns when `format` is fully processed. Returns `EOF` if end-of-file or error occurs before any conversion; otherwise, the number of items converted and assigned. Each of the arguments following `format` must be a pointer. Format string may contain:

- *Blanks*, *Tabs* : ignored
- *ordinary characters* : expected to match next non-white-space
- `%` : Conversion specification, consisting of `%`, optional assignment suppression character `*`, optional number indicating maximum field width, optional [`hlL`] indicating width of target, conversion character.

Conversion characters :

`d`
decimal integer; `int*` parameter required

`i`
integer; `int*` parameter required; decimal, octal or hex

`o`
octal integer; `int*` parameter required

`u`
unsigned decimal integer; `unsigned int*` parameter required

`x`
hexadecimal integer; `int*` parameter required

`c`
characters; `char*` parameter required; up to width; no `'\0'` added; no skip

`s`
string of non-white-space; `char*` parameter required; `'\0'` added

`e,f,g`
: floating-point number; `float*` parameter required

`p`
: pointer value; `void*` parameter required

`n`
: chars read so far; `int*` parameter required

`[...]`
: longest non-empty string from set; `char*` parameter required; `'\0'`

`[^..]`
: longest non-empty string not from set; `char*` parameter required; `'\0'`

`%`
: literal `%`; no assignment

`int scanf(const char* format, ...);`
: `scanf(f, ...)` is equivalent to `fscanf(stdin, f, ...)`

`int sscanf(char* s, const char* format, ...);`
: Like `fscanf()`, but input read from string `s`.

`int fgetc(FILE* stream);`
: Returns next character from stream `stream` as an `unsigned char`, or `EOF` on end-of-file or error.

`char* fgets(char* s, int n, FILE* stream);`
: Reads at most the next `n`-1 characters from stream `stream` into `s`, stopping if a newline is encountered (after copying the newline to `s`). `s` is `NULL`-terminated. Returns `s`, or `NULL` on end-of-file or error.

`int fputc(int c, FILE* stream);`
: Writes `c`, converted to `unsigned char`, to stream `stream`. Returns the character written, or `EOF` on error.

`char* fputs(const char* s, FILE* stream);`
: Writes `s`, which need not contain `'\n'` on stream stream. Returns non-negative on success, `EOF` on error.

`int getc(FILE* stream);`
: Equivalent to `fgetc()` except that it may be a macro.

`int getchar();`
: Equivalent to `getc(stdin)`.

`char* gets(char* s);`
: Reads next line from `stdin` into `s`. Replaces terminating newline with `'\0'`. Returns `s`, or `NULL` on end-of-file or error.

`int putc(int c, FILE* stream);`
: Equivalent to `fputc` except that it may be a macro.

`int putchar(int c);`
: `putchar(c)` is equivalent to `putc(c, stdout)`.

`int puts(const char* s);`
: Writes `s` and a newline to `stdout`. Returns non-negative on success, `EOF` on error.

`int unget(int c, FILE* stream);`
Pushes `c` (which must not be `EOF`), converted to unsigned char, onto stream `stream` such that it will be returned by the next read. Only one character of pushback is guaranteed for a stream. Returns `c`, or `EOF` on error.

`size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream);`
Reads at most `nobj` objects of size `size` from stream `stream` into `ptr`. Returns the number of objects read. `feof` and `ferror` must be used to determine status.

`size_t fwrite(const void* ptr, size_t size, size_t nobj, FILE* stream);`
Writes to stream `stream`, `nobj` objects of size `size` from array `ptr`. Returns the number of objects written (which will be less than `nobj` on error).

`int fseek(FILE* stream, long offset, int origin);`
Sets file position for stream `stream`. For a binary file, position is set to `offset` characters from `origin`, which may be `SEEK_SET` (beginning), `SEEK_CUR`(current position) or `SEEK_END` (end-of-file); for a text stream, `offset` must be zero or a value returned by `ftell` (in which case `origin` must be `SEEK_SET`). Returns non-zero on error.

`long ftell(FILE* stream);`
Returns current file position for stream `stream`, or `-1L` on error.

`void rewind(FILE* stream);`
`rewind(stream)` is equivalent to `fseek(stream, 0L, SEEK_SET); clearerr(stream)`.

`int fgetpos(FILE* stream, fpos_t* ptr);`
Assigns current position in stream `stream` to `*ptr`. Type `fpos_t` is suitable for recording such values. Returns non-zero on error.

`int fsetpos(FILE* stream, const fpos_t* ptr);`
Sets current position of stream `stream` to `*ptr`. Returns non-zero on error.

`void clearerr(FILE* stream);`
Clears the end-of-file and error indicators for stream `stream`.

`int feof(FILE* stream);`
Returns non-zero if end-of-file indicator for stream `stream` is set.

`int ferror(FILE* stream);`
Returns non-zero if error indicator for stream `stream` is set.

`void perror(const char* s);`
Prints `s` and implementation-defined error message corresponding to `errno`:
`fprintf(stderr, "%s: %s\n", s, "error message")`
See `strerror`.

B.10 `<stdlib.h>`

`double atof(const char* s);`
Returns numerical value of `s`. Equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char* s);`
Returns numerical value of `s`. Equivalent to `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char* s);`
Returns numerical value of `s`. Equivalent to `strtol(s, (char**)NULL, 10)`.

`double strtod(const char* s, char** endp);`

Converts prefix of `s` to `double`, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` if `endp` non-`NULL`. If answer would overflow, `HUGE_VAL` is returned with the appropriate sign; if underflow, zero returned. In either case, `errno` is set to `ERANGE`.

`long strtol(const char* s, char** endp, int base);`

Converts prefix of `s` to `long`, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` if `endp` non-`NULL`. If `base` between 2 and 36, that base used; if zero, leading `0X` or `0x` implies hexadecimal, leading `0`implies octal, otherwise decimal. Leading `0X` or `0x` permitted for base 16. If answer would overflow, `LONG_MAX` or `LONG_MIN` returned and `errno` is set to `ERANGE`.

`unsigned long strtoul(const char* s, char** endp, int base);`

As for `strtol` except result is `unsigned long` and error value is `ULONG_MAX`.

`int rand();`

Returns pseudo-random number in range `0` to `RAND_MAX`.

`void srand(unsigned int seed);`

Uses `seed` as seed for new sequence of pseudo-random numbers. Initial seed is `1`.

`void* calloc(size_t nobj, size_t size);`

Returns pointer to zero-initialised newly-allocated space for an array of `nobj` objects each of size `size`, or `NULL` if request cannot be satisfied.

`void* malloc(size_t size);`

Returns pointer to uninitialised newly-allocated space for an object of size `size`, or `NULL` if request cannot be satisfied.

`void* realloc(void* p, size_t size);`

Changes to `size` the size of the object to which `p` points. Contents unchanged to minimum of old and new sizes. If new size larger, new space is uninitialised. Returns ponter to the new space or, if request cannot be satisfied `NULL` leaving `p` unchanged.

`void free(void* p);`

Deallocats space to which `p` points. `p` must be `NULL`, in which case there is no effect, or a pointer returned by `calloc()`, `malloc()` or `realloc()`.

`void abort();`

Causes program to terminate abnormally, as if by `raise(SIGABRT)`.

`void exit(int status);`

Causes normal program termination. Functions installed using `atexit` are called in reverse order of registration, open files are flushed, open streams are closed and control is returned to environment. `status` is returned to environment in implementation-dependent manner. Zero indicates successful termination and the values `EXIT_SUCCESS` and `EXIT_FAILURE` may be used.

`int atexit(void (*fcm)(void));`

Registers `fcn` to be called when program terminates normally. Non-zero returned if registration cannot be made.

`int system(const char* s);`

Passes `s` to environment for execution. If `s` is `NULL`, non-zero returned if command processor exists, return value is implementation-dependent if `s` is non-`NULL`.

`char* getenv(const char* name);`
Returns (implementation-dependent) environment string associated with `name`, or `NULL` if no such string exists.

`void* bsearch(const void* key, const void* base, size_t n, size_t size, int (*cmp)(const void* keyval, const void* datum));`

Searches `base[0]...base[n-1]` for item matching `*key`. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater. The `n` items of `base` must be in ascending order. Returns a pointer to the matching entry or `NULL` if not found.

`void qsort(void* base, size_t n, size_t size, int (*cmp)(const void*, const void*));`

Arranges into ascending order the array `base[0]...base[n-1]` of objects of size `size`. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater.

`int abs(int n);`
Returns absolute value of `n`

`long labs(long n);`
Returns absolute value of `n`

`div_t div(int num, int denom);`
Returns in fields `quot` and `rem` of structure of type `div_t` the quotient and remainder of `num/denom`.

`ldiv_t ldiv(long num, long denom);`
Returns in fields `quot` and `rem` of structure of type `ldiv_t` the quotient and remainder of `num/denom`.

B.11 `<string.h>`

`char* strcpy(char* s, const char* ct);`
Copy `ct` to `s` including terminating `NULL`. Return `s`.

`char* strncpy(char* s, const char* ct, int n);`
Copy at most `n` characters of `ct` to `s`. Pad with `NULL`s if `ct` is of length less than `n`. Return `s`.

`char* strcat(char* s, const char* ct);`
Concatenate `ct` to `s`. Return `s`.

`char* strncat(char* s, const char* ct, int n);`
Concatenate at most `n` characters of `ct` to `s`. Terminate `s` with `NULL` and return it.

`int strcmp(const char* cs, const char* ct);`
Compare `cs` and `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`int strncmp(const char* cs, const char* ct, int n);`
Compare at most `n` characters of `cs` and `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`char* strchr(const char* cs, int c);`
Return pointer to first occurrence of `c` in `cs`, or `NULL` if not found.

`char* strrchr(const char* cs, int c);`
Return pointer to last occurrence of `c` in `cs`, or `NULL` if not found.

`size_t strspn(const char* cs, const char* ct);`
Return length of prefix of `cs` consisting entirely of characters in `ct`.

`size_t strcspn(const char* cs, const char* ct);`
Return length of prefix of `cs` consisting entirely of characters *not* in `ct`.

`char* strpbrk(const char* cs, const char* ct);`
Return pointer to first occurrence within `cs` of any character of `ct`, or `NULL` if not found.

`char* strstr(const char* cs, const char* ct);`
Return pointer to first occurrence of `ct` in `cs`, or `NULL` if not found.

`size_t strlen(const char* cs);`
Return length of `cs`.

`char* strerror(int n);`
Return pointer to implementation-defined string corresponding with error `n`.

`char* strtok(char* s, const char* t);`
A sequence of calls to `strtok` returns tokens from `s` delimted by a character in `ct`. Non-`NULL` `s` indicates the first call in a sequence. `ct` may differ on each call. Returns `NULL` when no such token found.

`void* memcpy(void* s, const void* ct, int n);`
Copy `n` characters from `ct` to `s`. Return `s`. Does not work correctly if objects overlap.

`void* memmove(void* s, const void* ct, int n);`
Copy `n` characters from `ct` to `s`. Return `s`. Works correctly even if objects overlap.

`int memcmp(const void* cs, const void* ct, int n);`
Compare first `n` characters of `cs` with `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`void* memchr(const char* cs, int c, int n);`
Return pointer to first occurrence of `c` in first `n` characters of `cs`, or `NULL` if not found.

`void* memset(char* s, int c, int n);`
Replace each of the first `n` characters of `s` by `c`. Return `s`.

B.12 `<time.h>`

`clock_t`
An arithmetic type representing time.

`CLOCKS_PER_SEC`
The number of `clock_t` units per second.

`time_t`
An arithmetic type representing time.

`struct tm`
Represents the components of calendar time:

`int tm_sec;`
seconds after the minute

`int tm_min;`
minutes after the hour

`int tm_hour;`
hours since midnight

`int tm_mday;`
day of the month

`int tm_mon;`
months since January

`int tm_year;`
years since 1900

`int tm_wday;`
days since Sunday

`int tm_yday;`
days since January 1

`int tm_isdst;`
Daylight Saving Time flag : is positive if DST is in effect, zero if not in effect, negative if information unavailable.

`clock_t clock();`
Returns processor time used by program or -1 if not available.

`time_t time(time_t* tp);`
Returns current calendar time or -1 if not available. If `tp` is non-`NULL`, return value is also assigned to `*tp`.

`double difftime(time_t time2, time_t time1);`
Returns the difference is seconds between `time2` and `time1`.

`time_t mktime(struct tm* tp);`
Returns the local time corresponding to `*tp`, or `-1` if it cannot be represented.

`char* asctime(const struct tm* tp);`
Returns the given time as a string of the form:
`Sun Jan   3 14:14:13 1988\n\0`

`char* ctime(const time_t* tp);`
Converts the given calendar time to a local time and returns the equivalent string. Equivalent to:
`asctime(localtime(tp))`

`struct tm* gmtime(const time_t* tp);`
Returns the given calendar time converted into Coordinated Universal Time, or `NULL` if not available.

`struct tm* localtime(const time_t* tp);`
Returns calendar time `tp` converted into local time.

`size_t strftime(char* s, size_t smax, const char* fmt, const struct tm* tp);`
Formats `*tp` into `s` according to `fmt`.

Notes:

- *Local* time may differ from *calendar* time, for example because of time zone.