## Contents

1 Introduction 7  
1.1 The Computing Course and Laboratory 7  
1.2 The C course 7  
1.3 The marking system and use of your logbook 7  
1.4 Hilary term 8  
1.5 The C language 8  
1.6 Typographical conventions 9  

2 Learning to use the system 11  
2.1 Logging in and getting started 11  
2.2 Applications you will be using 11  
2.3 The C development environment 12  
  2.3.1 Creating a new project 12  
  2.3.2 Compiling, running and debugging your program 13  
  2.3.3 Opening and closing existing programs 15  
2.4 Terminal and the Unix command line 15  
  2.4.1 Creating and manipulating directories 15  
  2.4.2 Manipulating files 16  
  2.4.3 Editing, compiling and running programs from the Unix shell 16  
  2.4.4 Command line compilation on the Mac using xcodebuild 16  
  2.4.5 Interaction between the command line and graphical environments 17  
2.5 Searching with Spotlight 17  
2.6 Logging out 17  

3 The elements of C 19  
3.1 Program Structure 19  
  3.1.1 Hello world 19  
  3.1.2 Code layout and style 20  
3.2 Variables 21  
  3.2.1 Types 21  
  3.2.2 Declaration and Names 22  
  3.2.3 Assignment 22  
3.3 Output using printf() 24  
3.4 Input using scanf() 26  
  3.4.1 Reading a single character 28  
3.5 Arithmetic Expressions and Operators 28  
  3.5.1 Type conversions and casts 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2</td>
<td>Some other operators</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Expressions, statements, and compound</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>statements</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>if statements</td>
<td>32</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Relational operators</td>
<td>32</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Logical operators and else-ifs</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Loops and Iteration</td>
<td>34</td>
</tr>
<tr>
<td>3.8.1</td>
<td>for loops</td>
<td>34</td>
</tr>
<tr>
<td>3.8.2</td>
<td>while loops</td>
<td>35</td>
</tr>
<tr>
<td>3.8.3</td>
<td>do...while loops</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>Mathematical functions</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Arrays</td>
<td>38</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Declaration</td>
<td>39</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Array elements</td>
<td>39</td>
</tr>
<tr>
<td>3.10.3</td>
<td>Initialisation</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>Functions</td>
<td>40</td>
</tr>
<tr>
<td>3.12</td>
<td>File I/O</td>
<td>42</td>
</tr>
<tr>
<td>3.12.1</td>
<td>Writing to Files</td>
<td>44</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Reading from Files</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Further C</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Random Numbers</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Complex numbers</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Multi-dimensional Arrays</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>String manipulation functions</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>switch statements</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>The ternary operator</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>More about variables</td>
<td>54</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Scope</td>
<td>54</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Global Variables</td>
<td>55</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Storage classes</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>The C Preprocessor</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Data structures</td>
<td>57</td>
</tr>
<tr>
<td>4.9.1</td>
<td>union</td>
<td>58</td>
</tr>
<tr>
<td>4.10</td>
<td>Enumerated types</td>
<td>59</td>
</tr>
<tr>
<td>4.11</td>
<td>Pointers</td>
<td>59</td>
</tr>
<tr>
<td>4.11.1</td>
<td>What is a pointer?</td>
<td>59</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Pointers and arrays</td>
<td>60</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Pointers and functions</td>
<td>60</td>
</tr>
<tr>
<td>4.11.4</td>
<td>Command line input</td>
<td>61</td>
</tr>
<tr>
<td>4.12</td>
<td>Custom variable types with typedef</td>
<td>61</td>
</tr>
<tr>
<td>4.13</td>
<td>Variable–argument functions</td>
<td>62</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Introduction to variable–argument functions</td>
<td>62</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Declaring a variable–argument function</td>
<td>62</td>
</tr>
<tr>
<td>4.13.3</td>
<td>Accessing the varargs list</td>
<td>62</td>
</tr>
<tr>
<td>A</td>
<td>Reserved Words and Program Filenames</td>
<td>65</td>
</tr>
</tbody>
</table>
Introduction

1.1 The Computing Course and Laboratory

Being able to program a computer is a vital skill for the modern physicist. The first year computing course is designed to teach computer programming in a scientific context, assuming no prior knowledge of computing. The programming language you will learn during this course is called C.

All undergraduates are required to attend the computing course in their first year. You will spend four half-day sessions in the Computing Lab during Michaelmas term. Sessions take place either in the morning (1000–1300hrs) or the afternoon (1400–1700hrs). People from either session may also work during the lunch-hour.

You will work singly, not in pairs as you will do in other parts of the Practical Course. Computing will contribute the equivalent of 4 days to your Practical Course record: 2 days in Michaelmas and 2 in Hilary. Be sure to check the Practical Course Handbook for more important information about computing and other labs, and the procedure and deadlines for completing practical work.

During your four sessions in the Computing Lab, you will be using Apple Macintosh computers, running the Mac OS X operating system which is a variant of Unix. Unix is a very powerful and flexible operating system and is preferred by many physicists and other technical persons for serious work above other, less capable systems. Mac OS X and Unix are described further in chapter 2.

1.2 The C course

In addition to acquainting yourself with the system, you will also learn the basics of computer programming in C. You will do this by working your way through to the end of Chapter 3 of this handbook, reading the explanations and attempting most of the exercises (some exercises have parts (a) and (b): in these cases (b) is optional and intended for students with previous programming experience or those wanting more of a challenge). If you have any trouble, please don’t hesitate to speak to a demonstrator: they are here to help.

We anticipate that you will finish the exercises in this book towards either the end of your second session or the beginning of your third. Don’t worry if you finish earlier or later. If you think you have finished the handbook then talk to a demonstrator who will assess your progress and then record your completion of the handbook’s exercises as CO00 (gaining you a day’s practical course credit). The demonstrator will then select one of the following problems for you to solve over the remaining sessions and write up as an Account:

CO11 Quadratic Equation; straight line fitting by least squares
CO12 Nuclear decay and the Doppler effect
CO13 Numerical Integration by Simpson’s rule and by a Monte Carlo method
CO14 Solution of non-linear equations; solution of a differential equation
CO15 Polynomial curve fitting by least squares
CO16 Graph plotting; Fourier Analysis

Section 1.3.4 of the Practical Course Handbook explains what to include in an Account.

1.3 The marking system and use of your logbook

You are encouraged to use your logbook whilst you are in the Computing Lab. Keeping a logbook whilst programming is just as important as when doing experimental work. When you have completed the programming problem (CO11–CO16), your Account will be marked by a demonstrator, who will expect you to explain in detail
how your program works; they will also test your program by running it. So it is in your own interests to keep a logbook as it will help you to explain your work to the demonstrator.

The kind of things that could be recorded in your logbook include:

- The outline design of your program;
- Steps you took which aren’t obvious from the problem script;
- Notes on any problems encountered as you tried to make your program work, and your solutions;
- Hints from demonstrators;
- Input data and output results, including graphs.

See also Section 1.3.2 of the Practical Course Handbook for more details on keeping a logbook.

1.4 Hilary term

In Hilary term you will be expected to solve a more challenging programming problem. By the end of Michaelmas Term you should have received an email informing you of which problem you will have to do. You can then collect the appropriate script from the foyer of the Practical Course. The problem must be completed and a Report written up and submitted to the Practical Course technicians’ office (DWB Room 206) by Friday noon, 6th week of Hilary Term. You will find an explanation of how to write a Report on our local web pages at http://www-teaching.physics.ox.ac.uk/computing/HowToWriteReports/reports.html.

The Computing Lab will be available for you to work on your problem throughout Hilary term. There will be a demonstrator present on Thursdays and Fridays to give assistance and advice to first years working on programming problems. First years will have priority on these days; other years nominally have priority on Mondays and Tuesdays. No-one has priority on Wednesdays when the Lab may be used freely. However, at all times, people wishing to do serious work have priority over those web-browsing or using email.

On Thursdays and Fridays in Hilary term access to the computers will be subject to availability rather than by formally-booked sessions. In practice, though, there are always likely to be machines available in the Computing Lab during Hilary.

You will also be free to use your own or college computing facilities, if you have access to them. Further advice about installing suitable software to allow you to program in C on your own computer will be made available in due course.

1.5 The C language

This course will only teach you a subset of the C programming language. Nevertheless, in doing the course you will acquire a solid foundation of skills required for programming in general, as well as those applicable to C itself.

C is a very powerful and versatile language which was originally developed in conjunction with the Unix operating system, around 1971 by Brian Kernighan and Dennis Ritchie. It was a derivative of the ‘B’ language, which was in turn a descendant of BCPL (Basic Combined Programming Language). C is still a popular and widely used language. For instance, many of the physicists working in the department use C (whether directly, or via a descendant such as C++) as their primary programming language. More importantly, though, its compact efficiency and versatility make C highly-suited not only to scientific programming, but also for writing systems software, applications and general-purpose programs. In addition to being a useful and popular language in its own right, C has also given rise to several derivative languages.

So another benefit of learning C is that a good foundation is gained for subsequently learning C++, Java, or indeed any of the derived languages.

1 The direct C derivatives include C++, Java, Objective-C and the new Microsoft language C#; C has also had an influence on more modern languages such as Perl and Python and it is intimately connected with Unix operating systems (including Linux and Mac OS X). Knowing about C therefore gives an added insight into many aspects of modern computing.

2 If you already know any C++ (or Java, or any other C derivative) please do not use any constructs that are not part of Standard C. This course deliberately only considers the current international standard (ISO) version of C. Therefore you should also not use any platform-specific extensions such as those found in Windows C compilers (eg. Turbo-C or Visual C).
C is neither the easiest language for a beginner to grasp, nor is it the most difficult. For those with previous programming experience, it should certainly be straightforward to learn. In any case, once it has been mastered, C constitutes an extremely powerful programming tool and one that is standard across a wide range of computing platforms. If you have a look at the hundreds of thousands of open source projects on websites such as http://freshmeat.net or http://sourceforge.net, you will see that the vast majority are written in C or one of its derivatives. C is typically the first (and sometimes the only) language made available on a platform: from the smallest embedded microprocessors, to Windows PCs and Macs, Linux and Unix workstations, right up to mainframes and supercomputers. Your efforts in learning the language will be well rewarded.

1.6 Typographical conventions

This handbook uses the following typographical conventions:

Most of the document appears in the Times font.

Typewriter font is used for anything typed, e.g. C code or Unix commands.

Sans serif is used for menu and application names

New terms and jargon are written in italics the first time they are used.
Learning to use the system

This chapter will introduce you to the facilities in the Computing Lab., which are mainly provided to complete the computing practicals but may also be used for other purposes, subject to availability and the local regulations. It is not necessary to work through this chapter in order to complete your first computing session, however you should read it before attempting the practicals because doing so will answer many basic questions about using the computers.

In the Computing Lab. you will be using Apple Macintosh computers running Mac OS X. OS X is based on the Unix operating system. Unix differs in several ways from Microsoft’s Windows environment; those of you who have used OS X before or a Linux variant have already used Unix and so may have encountered some of the “Unix-isms” that will be presented here. Physicists often prefer using Unix to Windows because, once you are familiar with it, it is a much more flexible and versatile environment for scientific and technical computing, as well as being generally regarded as more secure and reliable (it is not as prone to viruses as some popular systems!). OS X provides a graphical environment called Aqua which differs from the equivalent on other Unix (the X Window System or just X) but which is highly advanced and hosts many of the sorts of application you will be used to seeing on other systems. The X Window System is also available from within the OS X environment, and can be used to access graphical applications running on remote Unix systems as well as some programs which are local to the Mac.

2.1 Logging in and getting started

Sit at one of the Macs. If the screen is blank, move the mouse to re-awaken it. Click OK to view a summary of the computing regulations and then click Accept to proceed. You should now see a login panel: check that the Caps Lock is off and enter your username in lower case where indicated. Press tab, now enter your password and press return. Ask a demonstrator if you need help.

The login panel should disappear and, after a brief wait, be replaced with your desktop, and the Getting Started web guide. The guide is a basic tutorial on how to use a Mac computer from the perspective of a Microsoft Windows user. If you feel you are a seasoned OS X user, much of what is written will be familiar to you.

Once you have read through the welcome page (which should also contain instructions on how to stop it from displaying at every startup) and have familiarized yourself with the Mac OS X environment, continue.

2.2 Applications you will be using

All the computers in the computing laboratory have been customized to have all essential programs in the Dock which is positioned to the right of the screen. Particularly important programs are:

- **Safari**: the default web browser.
- **Terminal**: provides access to the Unix command line interface (see section 2.4).
- **X11**: launches the X Window System, which may be required to run certain other graphical applications. You need not concern yourself with this program right now. It will become important should you decide to try an astrophysics problem in your third year (and may also be required by other physics related software in the future).

---

1 People used to seeing the Dock at the bottom are free to change it back. The reason it has been moved is to create more space on widescreen computers, and to give some of the demonstrators a bit of nostalgia for another operating system: NeXTSTEP, on which OS X is based.
2.3 The C development environment

The environment in which you will be writing, compiling and running your C programs is called Xcode and can be found in the Dock. Xcode is the Integrated Development Environment or IDE that Apple provide for creating many varieties of projects for Mac OS X, including graphical applications, libraries and frameworks for use by other applications and web applications. Throughout the computing course you will be creating programs that interact through the so-called standard input and output mechanism, which are called “Tools” within Xcode.

Xcode makes it easy to manage projects that contain multiple files (these are described in Appendix B) and provides user–friendly interfaces to the compiler and debugger. Another way in which Xcode helps you to write C programs is that it can highlight C code such that it is easier to distinguish parts of the program, e.g. comments are in green, numbers in blue and so on. It also has functionality to inspect and check for matching brackets in your program; many errors can be tracked down to mismatching or unmatched brackets.

Xcode can be launched via its icon shown above, either by double–clicking on it in the Finder or by clicking once on its icon (shortcut) in the Dock if there is one.

2.3.1 Creating a new project

When Xcode is launched for the first time it will display a ‘Welcome to Xcode’ window. You can read the tutorials that it offers at your leisure, if you wish, though you may simply want to disable this launch window (uncheck ‘Show at launch’ towards the bottom left of this window). Thereafter it will not open any windows when launched, but the word nearest the Apple logo in the menu bar at the top left of the screen will change to Xcode. Ensure this is the case by clicking the Xcode icon in the Dock. From the ‘File’ menu in the menubar, select ‘New Project...’ and then ‘Practical Course Project’ from the list which appears, it is in the ‘Practical Course’ group. Click next, then you will be asked to choose a name for your new project. Do so; in this example we will use the name “hello”. You should not choose a name containing a dot ‘.’ as this conflicts with the use of ‘.’ to denote file types, like ‘picture.jpg’. Using a dot in your project name can confuse Xcode, making it difficult to examine the results of your project. Click on finish, and a window similar to figure 2.1 will appear. For each program you create on the course you should go through the steps of generating a new Xcode Standard Tool project and edit the main.c file presented within this new project. It is possible to add multiple files to a single project, but this will not compile them as separate programs so will produce unexpected behaviour. If you follow the instructions above and generate a new project for each new program, then you will have a single file ‘main.c’ containing your source code in each project. It is not a problem that they are all called main.c as they are stored separately, one per project. Keeping your projects separate in this way makes it easy to manage which code belongs with which program; if when you are more comfortable with the system you need to generate larger projects with multiple files it will be easy to extend this “one project per program” recipe to cover that.

The list on the right shows the files associated with the project – the only one that we are interested in is main.c which contains the C program. Double–click on it to open an editor window. You will see that it contains some sample code. Edit this code so that it looks like the following program, or if you prefer simply delete everything in the editor (select it all and press the Backspace key) and type the program in.

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

int main ()  
{
    printf("Hello, World!\n");
    exit(0);
}
```

Save the file by selecting ‘Save’ from the File menu. You are not prompted for a file name as Xcode has already named this file as main.c and created it within your project’s directory.

---

2This is not related to the X Window System, but someone at Apple likes the letter X!
1Even if you do not put a program into the Dock explicitly, any running application’s icon will appear in the Dock with a dot next to it.
4If you actually want multiple C files in one project, see Appendix B.
5The other two files are the ‘executable’ - the program compiled from your C code; and a template manual page (see section D).
2.3.2 Compiling, running and debugging your program

The compiler is a program which can translate source code, i.e. the C which you have written, into machine code which the computer can understand. The computer cannot understand source code directly, therefore every time you write a program (or edit an existing program) you must compile it before you can run it. During development you may find that your program contains errors or bugs and needs to be edited and recompiled; it is very common for the edit–compile–run–debug cycle to be followed many times before a working program is created. For more information on debugging C code, see Appendix C.

To compile a program select ‘Build’ from Xcode’s Build menu. If all is well, the message “Build succeeded” will be displayed in the editor window. If there are any errors, the message will be “Build failed” and a red X will appear in the bottom–right of the window. Clicking this will open another Build Results window (see figure 2.2) showing what the errors were – if you double–click on an error in this window then the editor window will be brought to the front, highlighting the line of your program where the error lies.

Sometimes a program can trigger warnings from the compiler, whether or not errors are also found. Warnings indicate code that the compiler can understand but which may be ambiguous or badly formed; they are indicated in Xcode by yellow triangles containing exclamation points. It is always wise to try and fix warnings as well as errors, as they may be symptomatic of code that although legitimate does not do what you might expect. In some workplaces the compiler is set up to treat warnings as errors and fail when it encounters them; therefore learning to resolve warnings now could save you some time in the future.

Once your code is clear of errors you may run the program. Select ‘Build and Run’ from the Build menu. Yet another window appears called the Run Log (figure 2.3 shows the output of the “Hello, World!” program above), showing the output of your program. If your program expects any input, it should be typed into the Run Log window. If your program does not exit, for instance if there is a bug, you can stop it by clicking the ‘Terminate’ button in this window. You will notice that when you run your program multiple times, the output is simply appended onto the Run Log. If this is confusing, select ‘Clear Logs’ from Xcode’s Debug menu or click the tiny white X at bottom–right of the Run log window to remove any previous output.

If you have not done so, enter the above program into an Xcode project and run it to get a sense of how to use the environment. If you have any problems please consult a demonstrator, as you will be using Xcode throughout the computing course.
Figure 2.2: Xcode’s build log, showing an example of an error found in C code.

Figure 2.3: Xcode’s run log, displaying the output of the simple program in section 2.3.1.
2.3.3 Opening and closing existing programs

You can open a previously-created Xcode project either using the ‘Open...’ item in the File menu or from ‘Recent Items’ in the File menu, depending on when you last visited this project. Note that using the ‘Open...’ item, you should open a file with the extension .xcodeproj because Xcode saves supporting information about your program. In the example above, we created a project called “hello” so Xcode created a folder also called hello, and within it a file called hello.xcodeproj. It is this file that should be opened to revisit the “hello” project. The easiest way to close a program when you have finished editing it is to select ‘Close Project’ from the File menu; if there are unsaved changes then you will be prompted to save them.

2.4 Terminal and the Unix command line

This section is intended to be used when you are comfortable with the system; skip it if you are just getting started as you can return later.

There will be occasions when Xcode is not available; for instance if you are using a Unix system that isn’t a Mac or if you have logged into a Mac remotely using an SSH client. You can log in like this to the course’s Mac server; SSH to ssh-teaching.physics.ox.ac.uk using your usual username and password. In both of these cases you can still edit, compile and run your program from the Unix command line, or shell. The Unix shell is a little like a DOS prompt in Microsoft’s Windows, although it is much more powerful making available hundreds or even thousands of commands, programs and utilities. There are much more detailed descriptions of the Unix shell environment, some of which are linked to from the course’s website http://www-teaching.physics.ox.ac.uk/computing/. If you are logged in at a Mac in the Computing Lab. then you can access the shell from the Terminal application; see section 2.2. Upon launching this application you will see a window with some text; the last line is similar to:

user@computer:~>

This is the shell’s prompt and is its way of indicating that it is ready to receive commands. Some basic information about Unix commands is available by typing help at this prompt – this is not a standard Unix command but is built in to the shell we use (which is called ‘bash’; the Bourne Again SHell). Most of this section describes commands which can be found on any Unix system, such as Linux or Solaris (both of which are used within the Oxford Physics department), although section 2.4.4 on xcodebuild is specific to Mac OS X. We cover xcodebuild because it can be run when using a Mac remotely, whereas Xcode itself cannot.

2.4.1 Creating and manipulating directories

It is useful to organise your work separating it out into different directories (also called folders on some systems); just as with filing paperwork into different physical folders it makes locating a particular file much easier. On a computer, it also makes it less likely that in saving a file with a particular name, you will overwrite an older file of the same name from a different project. When you first launch a Terminal window or first log in to the computer, your current working directory will be your home directory which is given the shorthand name “~”. The current working directory is the one that, by default, text editors and other programs will save files into. You can see what is in the current directory by giving the ls command. To create a new directory within this, use the mkdir command as in:

user@computer:~> mkdir new_directory

where you should replace new_directory with the name you wish to give the new directory. You can now change directory into this new one using cd like this:

user@computer:~> cd new_directory
user@computer:~/new_directory>

Observe how the shell prompt has changed to indicate your new working directory. You can get back to your home directory either by typing cd .. or just cd on its own. The special directory name “..” refers to the parent of the current working directory, while the command cd always returns you to your home directory. Finally, an empty directory can be removed using the rmdir command:

user@computer:~> rmdir new_directory
2.4.2 Manipulating files

Imagine you have created a file, useless.c with Xcode or a text editor and decide that you do not require it any more. The rm command – short for remove – can be used to delete it:

```
user@computer:˜> rm useless.c
```

rm can also be used to remove a directory containing files, which is something that rmdir cannot do. The syntax in this case is:

```
user@computer:˜> rm -rf somedir
```

You can copy a file using the cp command:

```
user@computer:˜/project1> cp myfile.c ../project2/
```

and move a file (similar to a copy except that the original is deleted) using mv:

```
user@computer:˜> mv oldproject/somefile.c newproject/
```

If you want to experiment with any of these commands, you can use the directory listing (ls) or even the Finder application to verify the results.

2.4.3 Editing, compiling and running programs from the Unix shell

Unix systems usually contain a number of text editors which can be used to edit or create C programs, which are after all just plain text files. A good editor that is suitable for users unfamiliar to Unix is pico; those with some familiarity with Unix may wish to try out emacs or vi which are harder to learn but more powerful.

Any of the editors listed above may be invoked by typing its name at the shell prompt, optionally followed by a filename to cause it to open that file when it starts; for example pico or pico hello.c. Unix systems are almost exclusively case-sensitive, so that typing any of Pico, PICO or pIcO will not work. This should be remembered when trying any of the commands listed in this section.

Once you have saved your program and quit the text editor (by pressing ctrl, the control key, and X together, if you are using pico), you can compile it using the GNU C compiler, which is invoked via the name gcc. For compiling programs such as those you will be creating on the computing course, a shell command such as:

```
gcc -Wall filename.c -o filename
```

should be used. The various options, also called arguments, passed to gcc here tell it to display all warnings, to read a file filename.c and create an executable program called filename. On some Unix systems, you may also need to provide -lm in order to gain access to the mathematical functions like sin() and friends. This is not necessary on Mac OS X.

Any errors or warnings your program generated will be displayed in the terminal window before the prompt reappears. If your program generated neither warnings nor errors then GCC will not display any output at all and the shell prompt will simply reappear. You can now launch your program by entering “./” followed by its filename at the prompt, e.g. ./filename

2.4.4 Command line compilation on the Mac using xcodebuild

As well as providing the graphical Xcode environment, the Mac OS X developer tools include xcodebuild which allows you to easily deal with Xcode projects from the Unix shell, e.g. if connected to a Mac remotely. Taking the “Hello, World!” project above as our working example, when Xcode was told to create a project named “hello” it created a directory also called “hello” in which to store its files. To use xcodebuild you must first change into this directory, which is done by typing cd hello at the shell prompt. Note that the prompt now changes to:

```
```

6It is recommended that you do not use word processing packages such as OpenOffice.org or Word to create your C programs, as they save formatting information in the file which will cause the compiler to fail.

7Users familiar with Unix or Linux will already be aware that emacs is the One True Editor and that all others pale in comparison.
to remind you that you are in a different directory. Getting back to the default, or “home” directory at any later point can be done by typing $cd$ on its own. From within this project directory, simply type $xcodebuild$ and the system will work out what it needs to do to compile your program. Output such as:

```
--- BUILDING NATIVE TARGET hello WITHOUT USING ANY BUILD STYLE ---
** BUILD SUCCEEDED **
```

indicates that there were no problems and $xcodebuild$ has compiled your program. By default, Xcode and $xcodebuild$ put the resulting executable into a directory inside the project directory called “build/Debug”, so you can launch your program by typing $build/Debug/hello$ at the prompt. Note that if your build was unsuccessful then the output will end like this:

```
** BUILD FAILED **
```

and the executable will not have been created. Compiler errors will be displayed within the $xcodebuild$ output, you should try an correct them and rebuild your project.

### 2.4.5 Interaction between the command line and graphical environments

It is possible to use the OS X pasteboard from within the terminal, so that data can be shared between graphical applications and the Unix command line environment. As well as the usual cut, copy and paste menu items, OS X provides the $pbcopy$ and $pbpaste$ commands. The first, $pbcopy$, takes text from its input and puts it on the pasteboard; $pbpaste$ prints the contents of the pasteboard to standard output. If you have created a program in Xcode that generates a lot of output to the Run Log, you could select and copy this output then in a terminal type $pbpaste > output.data$ to put the results into a file. More experienced Unix users will be aware of command piping; the $pbcopy$ and $pbpaste$ commands can be used in a pipeline.

### 2.5 Searching with Spotlight

Spotlight is a powerful search feature included with Mac OS X. When you edit a file – whether in the graphical environment or at the command line – information about it is imported into a database allowing you quickly to locate it again. Consider this example: you have been working on a practical related to the Ising model, and need to find the files associated with this practical. Click on the magnifying–glass icon in the top right corner of the screen, and type “Ising” into the field. A list of results will appear under the Spotlight field. You can open any of these files by clicking on it, or press Enter to bring up a results window in which you can refine your search. Spotlight can also be invoked from the command line using commands such as $mdls$ and $mdfind$.

### 2.6 Logging out

To log out of the Mac when you have finished your session, select “Logout Your Name” from the Apple menu (which you get by clicking the Apple logo in the menu bar) and then answer the dialogue that appears. If you have any unsaved work, the computer may prompt you whether to save it before actually logging you out. You should log out after you have finished using the system to ensure that other people may use the computer and to stop others from having access to your sensitive data such as e–mail or files.
The elements of C

These notes give a basic introduction to programming using C and should contain all the information you need for the first-year course. Since C is such a popular and widely-used programming language there are many sources of information about it, both online and in book form. A list of online C information sources is available on the first year course web page, and a bibliography is provided in Appendix D.

3.1 Program Structure

3.1.1 Hello world

You have already written, compiled and run a simple C program:

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

int main()
{
    printf("Hello world\n");
    exit(0);
}
```

Though very short, this requires some explanation. C programs are organised into structural units known as functions (see section 3.11) and `main()` is one such function. In fact the `main()` function is the only one necessary for a C program to run; for now take it as read that the line `int main()` and the curly braces are necessary parts of a C program. The code you write should go between the curly braces, the Xcode project template indicates this for you. As you proceed through this handbook you will learn more about the structure of C programs.

The program starts at the first line after the opening brace in `main()` and statements are executed sequentially until the closing brace is reached, signalling the end of the `main()` function. When this happens the Operating System cleans up any space and resources used by your program; it has exited.

The first two lines,

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

contain a special kind of statement known as a preprocessor directive (described in more detail in section 4.8) which make certain predefined functions (also known as library functions) available to your program. These two libraries are so significant and the things they contain so frequently used that you should always include these two lines in your programs.

Moving on to the first line of code inside the `main()` function: `printf()` is a library function which is used to print formatted output to the screen. Once a function has been defined it is used, or called, by typing its name followed by parentheses. Pieces of information, known as arguments, may be passed to a function – this is done by placing the arguments inside the parentheses.

In this case we created a sequence or string of characters by enclosing them in double-quotes, and told `printf()` that we wanted it to print them by passing the string as an argument. The “\n” is known as an escape sequence and is used to indicate a newline character. Escape sequences give you the ability to represent ‘invisible’ characters such as the newline or tab in a string, giving great control over the output. It is not valid C to put a “real” new line in a string, so \n is used to represent one. `printf()` will be used frequently through the course, and is described in more detail in Section 3.3.

Finally, rather than just letting the program end by reaching `main()`’s closing brace, `, a call to the standard library function `exit()` has been used. Calling `exit()` means that the system does a bit of tidying up and

19
returns the *exit code* it is passed (in this case 0) to whatever ran the program. `exit()` may be called from anywhere within a C program, and provides a useful way of quitting a program that has encountered an error or other problem. It also allows your program a means of reporting whether or not its run was successful: by convention exit code 0 means “successful completion”; non-zero codes (such as 1) usually denote an abnormal termination of some sort, but you can choose whichever values you wish to use and assign them whatever meaning you like.\(^1\) The preprocessor constants (see section 4.8) `EXIT_SUCCESS` and `EXIT_FAILURE` are defined so that whatever system you run your code on it exits with a sensible value; e.g. `exit(EXIT_SUCCESS)` will emit 0 on a Unix system and whatever is appropriate on others.

Notice also the punctuation: semi-colons, “;”, are used to terminate each *statement* within the function. This does not mean that *every* line of a C program must end in a semi-colon (for example it would be a mistake to terminate the line “```int main()`” with one). By following the examples in this handbook, and with practice it will become apparent when they are required; forgetting them is a common mistake made by beginners, so don’t get too frustrated if you do.

```
Exercise 3.1
Modify your `hello.c` program to print out a short message, perhaps “Hello” followed by your name.
```

If you get errors when you try to compile or run a program, then check it carefully for small mistakes like missing punctuation marks (especially the “;”s on the ends of lines). Even seemingly trivial mistakes can cause big problems in computer programs!

### 3.1.2 Code layout and style

C is a *free-format* language which means that you have great freedom over the layout of your code. *White space* (ie. spaces and tabs) are generally ignored by the compiler, and even newlines are mostly\(^2\) ignored. In particular, you have the freedom to insert blank lines to space out logical sections of your code and to split long statements over several lines. It makes code infinitely easier to read if you use spaces or tabs to indent blocks of statements within braces `{}`. In practice we recommend 2 to 4 spaces, because tabs can end up pushing you too far over to the right. As mentioned in Chapter 2, Xcode will automatically help you with your code layout, including indentation. The demonstrators can also guide you on good practice.

As an example of how *unimportant* layout is to the compiler, the *Hello world* program could perfectly legally be written as follows (the C preprocessor statements are not free-format: they must still be written one per line):

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

int
main(){printf
("Hello world - this program is a complete mess!\n"
);exit
( 0)
;}
```

As you can see from the chaos above, whilst layout is mostly irrelevant to the compiler, it is vital for humans! It can and must be used to emphasize the structure of your programs to enhance their readability and improve understanding. Bad layout will often lead to incomprehensible code and hence increase the number of mistakes likely to be made: especially if you (or someone else) subsequently has to modify your program. It cannot be over-emphasized how important it is to get into the habit of using good layout: if acquired as a beginner it will be a habit that will remain with you always and help to make you a better programmer.

\(^1\)It is sensible to stick to 0 for success unless you have good reason not to, because it is such a strongly-used convention.

\(^2\)There are a few instances in C where newlines are important: eg. they are illegal within strings (as already mentioned); they are also required to terminate pre-processor statements.
Ultimately, though, code layout is a stylistic aspect of programming and so to an extent a matter for personal taste. C’s exceptionally free-format has lead to a variety of different styles for laying out programs; however the good ones are always clear and logical. The examples in this handbook (apart from the one above!) demonstrate a particularly clear and simple style which we recommend. You may follow another recognised style as long as it is clear and you are consistent.

3.2 Variables

Variables are named places in the computer’s memory that are used to store data. Conceptually they are much like variables in algebra and so you should find them fairly intuitive. However, unlike mathematical variables, C (and many other programming languages) requires you to declare variables before you use them, by stating their name and the type of data they will hold.

To draw an analogy, you can think of variables as named boxes. The name of the variable (e.g. \(x\) or \(radius\)) tells us which box to look at. The current numerical value of that variable is stored inside the box. Having named variables makes it easy to keep track of the various numbers that are typically used in calculations.

3.2.1 Types

There are several different types of variable which occupy different amounts of memory and which can store different kinds of data. The most important types are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind of data stored — amount of memory occupied</th>
<th>Example values</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integers (ie. whole numbers) in the range -2147483648 to 2147483647 — 4 bytes (32 bits)</td>
<td>-123456789, 0, 1, 17, -23, 1000000</td>
</tr>
<tr>
<td>float</td>
<td>Real numbers stored to only about 6 significant figures in the range of about (1.2 \times 10^{-38}) to (3.4 \times 10^{38}) — 4 bytes (32 bits)</td>
<td>0.0013, 3.14159, 3e+8, 1.6e-19, -2.0</td>
</tr>
<tr>
<td>double</td>
<td>Real numbers stored to 15 significant figures in the range of about (2.2 \times 10^{-308}) to (1.79 \times 10^{308}) — 8 bytes (64 bits)</td>
<td>0.0, -1.087e-102, -2.0, 3.141592653589</td>
</tr>
<tr>
<td>char</td>
<td>A special integer type with a range of -128 to 127. Hold the numeric code of a single text character — 1 byte (8 bits)</td>
<td>'a', 'A', 'z', '0', '9', '?', ';', '\n', (9), -1, 127</td>
</tr>
</tbody>
</table>

Some of the examples in the table use exponential notation. This is very straightforward: for example “3e8” means \(3 \times 10^8\) and “-1.087e-102” means \(-1.087 \times 10^{-102}\).

The sizes and ranges of the different types may vary depending on the computer system you’re using. However, chars are usually 8 bits and are guaranteed to be able to store values from 0 to 127 (they aren’t guaranteed to store negative values, though they usually do in practice). In addition, ints are almost always 4 bytes and doubles 8 bytes.

Floats (so-called because they implement floating-point arithmetic) are too imprecise for most scientific computing and hence doubles (which are double-precision floats) should nearly always be used instead. We mention the float type simply to provide context for the double; in your code try to remember to use doubles. The float type has a precision of 6 digits, whereas the precision of a double is 15 digits.

The double type can be less accurate than ints. Although doubles can store fractional values and hence store numbers to greater precision, ints store exact values (though they are limited to whole numbers). As an example of the difference between accuracy and precision: “\(g=10\)” is less precise than “\(g=8.80665324\)” but it is a much more accurate value of the mean acceleration due to gravity at the Earth’s surface. Even better, “59.998” is more precise than “60” but it is less accurate as a count of the number of seconds in a minute!

The values stored by chars are integer numbers; however, because chars are usually used to store the numeric codes of characters, character constants like ‘A’, ‘5’ or ‘\n’ are normally used with them. For instance, the code on this system\(^3\) for the letter ‘a’ is 97, the digit ‘1’ is 49, and the newline character ‘\n’ is

\(^3\)If you want to see a full list of the ASCII character codes used by this system, type the Unix command: \texttt{man ascii}
10. Be careful not to confuse the character ‘2’ with the number 2, as they have quite distinct meanings. The numeric values of character codes are unmemorable and may vary from system to system, so it is always best to use character constants; their meaning is also more transparent.

Note that forward-quotes, ’ ’, delimit character constants. Double quotes, " " , are used to define strings (which are sequences of characters – chars can only store individual characters): do not confuse the two! We have already encountered strings with printf(), in fact they were string constants.

"Here is a string constant"

There are several other types available in C including short and long ints and long doubles which have varying sizes and ranges; and there are also unsigned versions of all the integer types (which can only store positive values). However, you can safely restrict yourself to ints, doubles, and chars for this course.

3.2.2 Declaration and Names

C requires you to declare all variables before they are used, so that the right amount of space can be allocated for them by the compiler and so that you have a record of the variable names you have used. In C a declaration specifies the type of variable and then one or more variable names (if there are multiple names they are comma-separated). Here are some examples of valid declarations:

```c
int a;
int i, j, level;
double x, y3, voltage, epsilon0;
char c, answer;
double hubble_constant;
```

Variable names must start with a letter (the underscore character, “_”, counts as a letter; variable names starting with an underscore are usually reserved for system use) and then may consist only of alphanumeric characters (i.e. letters and numbers). Spaces and punctuation characters are not allowed. If you are a beginner you will avoid making mistakes if you restrict yourself to using lower-case for the names of your variables (and functions). See also Section 2.4.3 on case-sensitivity.

Where relevant you should give your variables a descriptive name, such as time, height, or z0. If you are coding a formula, try to match the letters or symbol-names used for the algebraic variables to the variable names you choose in your program. There are various reserved words which you cannot use for variable names because C uses them for other things; these are listed in Appendix A.

In older versions of the C standard it was necessary for all variable declarations to be grouped together and to appear at the beginning of a block of code before any other statements, i.e. after a { . While this is no longer the case with the latest version of the standard (known as C99), it is still good practice to do so. Grouping variable declarations together makes it easier to see what variables are used and where they were created.

3.2.3 Assignment

Data is stored in a variable by assigning a value to the variable. Assignments are made by putting the variable name on the left, followed by a single =, followed by what is to be stored. eg.

```c
x = 5;
```

Referring back to our named box analogy, the statement above identifies the value (5) we want to store and the name of the box (‘x’) in which to store it.

In addition to simple constants (such as 5, 2.6, or ‘a’) the thing being stored may be the value of another variable or a more complicated expression (we will say more about arithmetic expressions in Section 3.5).

```c
y = x + 4;
```
You also have the option of assigning variables an initial value when you declare them (although in these cases, you can usually only assign constants); eg.

```c
int p=3, q=4;
double sum=0.0;
char c='y';
```

Note that although 0 is an integer, by writing it above as 0.0 we indicate that it is a double constant: there are certain situations where it can be important to be explicit in this way.

There are some differences between the syntax of C and normal algebra which are important. Assignment statements work from right to left only: `x = 5` is fine, but `5 = x` doesn’t make sense and will cause an assignment error. If you like, you can think of the equals sign as an arrow pointing from the value on the right, to the variable name on the left: `x ← 5` and read the expression as “assign 5 to `x`” (or, if you prefer, as “`x` becomes 5”). However, C will still accept many expressions that might be used in algebra, although in some cases the exact meanings may differ:

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

The assignment above reads as: “assign 0 to c, assign c to b, assign b to a”.

There are also statements that are algebraically nonsense, that are perfectly valid in C (and indeed in most other programming languages). The most common example is incrementing a variable:

```c
i = 2;
i = i + 1;
```

The second line in this example is nonsensical in maths, but makes sense in C if you think of the equals as an arrow pointing from right to left. To describe the second statement in words: set the new value of `i` to be its current value plus one. We look in the `i` box for the current value, and one to that, and store it back in `i`. In this way `i` is incremented from 2 to 3. Having described what variables are and how they work we’re now going to have a look at them in action.

---

**Exercise 3.2-1**

Type in, compile and run the following example program.

```c
/*
 * A comment to explain that this is a simple program to add up two numbers */
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int i, j, sum; /* declare 3 variables */
    /* Assign initial values to variables */
    i = 3;
    j = 5;
    /*Add them up */
    sum = i+j;
    //Output the results
    printf("i is %d and j is %d\n", i, j);
    printf("Their sum is %d\n", sum);
    exit(0);
}
```

**Remember:** create a new Xcode project for each program.

---

“The syntax of a language refers to its grammar and structure: ie. the order and way in which things are written.
Now do exercise 3.2-1. Compile and run the program. It should be fairly obvious what it does, though perhaps not what every bit of the code means. What would have been immediately apparent is that whenever we referred to a variable in the program, C replaced its name with the value we’d assigned to it.

You will also have noticed a new feature in the program not related to variables: the use of comments. If the compiler sees the character pair “/\*” it completely ignores everything that follows until the characters “\*/” are read. (The only exception is that comments cannot occur inside strings, other comments or character constants: in those cases “/\*” has no special meaning whatsoever). A second style of comment introduced in the C99 standard starts with “//” and runs until the end of the line. The example shows several different ways in which comments can be used: as with code layout they have no significance to the compiler and how you insert them is largely a matter of personal taste—just ensure they are clearly placed. Comments should be liberally used throughout programs to explain what the code is doing, but keep them succinct.

We have also used printf() in a new way to output not just text but the values stored in our int variables. The use of printf() is described further in the next section.

**Exercise 3.2-2**

(a) Change the program so it subtracts the two numbers, rather than adds them up. Try using different numbers.

(b) Experiment with the program to prove to yourself that C is case-sensitive; eg. check whether a variable named sum is the same as one called SUM. You could also try capitalising the names of one of the functions: this may cause some strange problems!

### 3.3 Output using printf()

Computer programs generally interact with the user or, in some cases, with other programs. These interactions occur as a result of inputs and outputs. The most common form of output (which we’ve already encountered) involves printing things to the screen; however, as we shall see later, it can also involve writing data to files, sending plots to a printer, etc.

We have already seen that the printf() function can print more than just a text message. The last example program included:

```c
printf("i is %d and j is %d\n", i, j);  
printf("Their sum is %d\n", sum); 
```

When you ran the program you’ll have noticed that the %d’s got replaced in the output by the values of i, j, and sum. printf() always requires a string (called the format string) as its first argument, and it may then be followed by one or more additional arguments (which can be variables, constants, or expressions) that you want it to format and print for you. The format string can contain three kinds of text: ordinary characters which are printed exactly as they appear; escape sequences for special characters like ‘\n’ for newline; and conversion specifications which begin with a ‘%’ and end with a conversion character such as ‘\d’.

There are several escape sequences defined in C (we have already encountered \n several times). The most useful ones are:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>&quot;</td>
<td>backslash itself</td>
</tr>
<tr>
<td>'</td>
<td>quote “’”</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote “”</td>
</tr>
</tbody>
</table>

For example,

```c
printf("\t\"My name is "s\\\"\n", "Jim");
```

Which prints

"My name is ‘Jim’"

Escape sequences may be used in strings or in character constants, eg. "\%d\t\%d" or '\t', and they count as a single character.

A conversion specification like "\%d" tells printf() to insert the value of the next argument at that point in the output: how the value is printed depends on which conversion character is used: eg. "\%d" means format an integer value as a decimal number. Here is an example:

```c
int a=4;
double x=2.5;
printf("Here is an integer: %d\t", a);
printf("and here is a double: %f\n", x);
```

Note that we used %f for the double: different conversion codes work with different types of value (see the table on the next page). You must be careful to match the type of the argument to an appropriate conversion. For example, you will get a strange result if you use %d with a double instead of an int! Our C compiler will warn you if you get the types wrong, but it doesn’t actually prevent you from doing it and will still compile your program. If the compiler gives out a warning in this way, you should not ignore it as while your code has still compiled, there is probably a problem with it which will manifest itself when you run the program.

The example code above would produce the following output:

```
Here is an integer: 4 and here is a double: 2.500000
```

You can see that, as requested, we got a tab after the 4, but the value of x has been printed to 6 decimal places: this is simply because %f always prints to 6 decimal places by default. You can override this default precision by using %.Nf where N is the number of decimal places you want. For example, using the same values of a and x, here’s another version of the example modified to specify the precision (its output is shown beneath it):

```c
printf("Here is an integer: %d\t", a);
printf("and here is a double: %.2f\n", x);
```

```
Here is an integer: 4 and here is a double: 2.50
```

In addition to specifying the precision, you can also specify a field-width: ie. the minimum number of characters that will be printed by a conversion. If there are fewer characters, spaces are usually used to pad out to the field-width; if there are more, then the field-width is exceeded. Be aware that the width includes any decimal point, + or - signs, and the letter "e" (or "E") if using an exponential format (eg. "-4.1e+03" occupies 8 characters). Field-widths can be used with all the conversion codes. They are essential for ensuring that columns are aligned when outputting tables of data. See the following example and its output:

```c
int a=1;
double x=-0.004;
printf("%3d\t%8.2f\n", a, x);
a=10;
x=-40.009;
printf("%3d\t%8.2f\n", a, x);
a=100;
x=-4001;
printf("%3d\t%8.2f\n", a, x);
```

```
1 -0.00
10 -40.01
100 -4001.00
```

---

If the field-width is written with a leading 0 (eg. %03d) then zeros are used for padding instead of spaces.

3.3. Output using printf() 25
The padding is normally placed to the left of the characters printed, so that they line up on the right (right-justification). You can get left-justification if you use a minus sign: %−3d, %−8.2f, etc. Notice also that printf() has rounded −40.009 to the correct precision, ie. −40.01.

Below is a table of the common printf() conversion codes. Note that the codes which expect an int will work with any integer type, including chars. However, they will give very strange results if you pass them a floating point type. Likewise, the double codes will happily accept a float but may produce strange output if you pass them an integer type.

<table>
<thead>
<tr>
<th>Conversion code</th>
<th>Argument type</th>
<th>Formatted as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>int</td>
<td>Ordinary number (eg. 365)</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>Equivalent to %d</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>Prints the character with this numeric code (see below)</td>
</tr>
<tr>
<td>%f</td>
<td>double</td>
<td>Real number, including a decimal point, (by default)</td>
</tr>
<tr>
<td>%e, %E</td>
<td>double</td>
<td>a precision of 6 decimal places (eg. 365.256000)</td>
</tr>
<tr>
<td>%g, %G</td>
<td>double</td>
<td>Alternative to %f which formats using exponential notation</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
<td>eg. 3.652560e+02, or 3.652560E+02</td>
</tr>
<tr>
<td>%</td>
<td></td>
<td>An intelligent combination of %f and %e; it tries to choose the more sensible format.</td>
</tr>
<tr>
<td>%</td>
<td></td>
<td>Print a string.</td>
</tr>
<tr>
<td>%</td>
<td></td>
<td>This is how you output the “%” character itself</td>
</tr>
</tbody>
</table>

There is no simple string type in C; for now it’s enough to know that you may print a string constant with printf() by using %s.

Three more examples:

```c
char c='A';
printf("%s %c is %d\n", "The ASCII code for", c, c);
```

The ASCII code for A is 65

```c
double x=365.256;
printf("%.1f\t%8.1e\t%.10E\n", x, x, x);
```

---

365.3 3.7e+02 3.6525600000E+02

```c
double y=1.25e-5;
printf("%12.4f %8.4g\n%12.4f %8.4g\n", y, y, y*1000, y*1000);
printf("%12.4f %8.4g\n%12.4f %8.4g\n", y*1000, y*1000, y*1e8, y*1e8);
printf("%12.4f %8.4g\n%12.4f %8.4g\n", y*1e8, y*1e8, y*1e12, y*1e12);
```

---

0.0000 1.25e-05
0.0125 0.0125
1250.0000 1250
12500000.0000 1.25e+07

It is up to you to ensure that the number of arguments after the format string matches the number of conversion specifications: compilers react differently to such mismatches, some will produce warnings others will not, some will compile and allow you to run your program where bizarre values will get printed. In all cases it is a mistake to have a different number of arguments and conversion specifications.

### 3.4 Input using scanf()

So far, we have only output data. This section describes how to accept input data using scanf(). scanf() can be thought of as the counterpart to printf(), working much like it in reverse: it inputs values into variables.
rather than outputting the values from variables. Like `printf()`, it expects a format string as its first argument, and then a series of arguments which should be references to variables into which values are to be input.

A `scanf()` format string looks essentially identical to a `printf()` one. The two differences are that:

- Spaces in the format string match any number of spaces and tabs in the input;
- The conversion specifications for floats and doubles are not quite the same: floats still use `%f`, but doubles must use `%lf` [Note: this is the letter ell, `l`, not the digit one, `1`; think of `%lf` as "long float"].

As you should mainly be using the `double` type, get used to writing `%lf` in `scanf()` statements.

Rather than use the ordinary name of the variable, you must prefix it with the `&` operator which gives a reference to the variable rather than the contents of the variable. `scanf()` needs to be able to write new values into variables, so it needs to know where the variable is, rather than what value it presently contains; it needs to know which box to store its result in, but doesn’t care what is currently in that box.

An example of using `scanf()` to input an x-y pair consisting of an `int` and a `double`:

```
Values input are:
4 0.00451
-------------------
/* Code fragment to use to input data of the above format */

int x;
double y;
scanf("%d %lf", &x, &y);
```

The exception to this is strings (ie. arrays of characters). You will encounter character arrays later in the course, for now remember that they do not require the `&` operator. Those who are interested in seeing why can consult section 4.11.2.

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

int main()
{
  int i;
  float f;
  double d;
  char strng[21];

  printf("type in an int: ");
  scanf("%d", &i);
  printf("value entered: %d\n\n", i);

  printf("type in a float: ");
  scanf("%f", &f);
  printf("value entered: %f\n\n", f);

  printf("type in a double: ");
  scanf("%lf", &d);
  printf("value entered: %f\n\n", d);

  printf("type in a string (up to 20 characters long): ");
  scanf("%s", strng);  /***Strings do NOT require the & operator***/
  printf("string entered: %s\n\n", strng);

  exit(0);
}
```
3.4.1 Reading a single character

`scanf()` ignores blanks and tabs in its format string, it also skips over white spaces (blanks, tabs and newlines) as it looks for input values. You do not need to explicitly put whitespace in your format string, except if you are trying to read a char – the seemingly obvious solution:

```c
scanf("%c",&char_variable);
```

may not work as you expect.

Normally, when you are asked for keyboard input you indicate to the system that you have finished typing by pressing the Return (or Enter) key. This appends a newline character to the characters you typed, and everything is placed by the system into what is called a buffer. The system then makes this input buffer available to the program, which in the above case is using `scanf()` to read one character from the buffer placing it in a variable called `char_variable`. However, an unfortunate quirk of using `scanf()` to read a single character in this way is that the newline character that was used to indicate the end of keyboard input is also left in the input buffer! (Note that this only happens when using a `%c` conversion). The next time you use `scanf()` the buffer is not empty, as it still has a newline character in it, which precedes any subsequent input placed in the buffer by the next invocation of `scanf()`. This subsequent invocation will try to read its values from a seemingly blank line, which could cause trouble (especially if you next try to read another char, as the value “‘n’ will be stored).

Happily, there is a simple way to work around this issue: you can force `scanf()` to read and discard any unwanted whitespace (including newline characters) prior to a `%c` conversion by using a leading space. ie. Instead of "%c" use " %c".

3.5 Arithmetic Expressions and Operators

The programs you write will nearly always involve numerical calculations. The rules for arithmetic expressions are largely common sense: anything that looks right mathematically usually is, with some minor changes (for example, the multiplication operator must always be included explicitly). In fact, C performs arithmetic in much the same way as an electronic calculator. Basic arithmetic calculations are expressed using C’s mostly obvious arithmetic operators.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Remainder</td>
<td>%</td>
</tr>
<tr>
<td>Grouping</td>
<td>( )</td>
</tr>
</tbody>
</table>

C generally evaluates expressions from left to right, but things enclosed in brackets are calculated first, followed by multiplications and divisions, followed by additions and subtractions. If in doubt use parentheses around the expressions you want evaluated first. As you can see below, parentheses may be nested, in which case the innermost expressions are evaluated first.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+3*4</td>
<td>14</td>
<td>Multiplication (and division) are evaluated before addition (or subtraction)</td>
</tr>
<tr>
<td>(2+3)*4</td>
<td>20</td>
<td>Parentheses over-ride normal operator precedence</td>
</tr>
<tr>
<td>2+(3*4)</td>
<td>14</td>
<td>The same as the default behaviour</td>
</tr>
<tr>
<td>6/2*3</td>
<td>9</td>
<td>Equal precedence operators are evaluated strictly left-to-right</td>
</tr>
<tr>
<td>14%5</td>
<td>4</td>
<td>The remainder of 14/5</td>
</tr>
<tr>
<td>4*(6/(3-1))</td>
<td>12</td>
<td>Innermost parentheses are evaluated first</td>
</tr>
<tr>
<td>1/2</td>
<td>0</td>
<td>Integer divisions discard fractional parts</td>
</tr>
<tr>
<td>1.0/2</td>
<td>0.5</td>
<td>The presence of a non-integer constant or variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>causes an expression to be calculated using floating-point arithmetic</td>
</tr>
</tbody>
</table>

*More experienced students may like to know that the buffer may alternatively be cleared, or flushed, using the `fflush()` library function.
Unlike some other programming languages, there is no built-in power operator to express $x^y$. For raising $x$ to small integer powers, the most effective method is to use $x \times x$, or $x \times x \times x$, etc. Otherwise, the library function \texttt{pow}(x,y) is available: it expects two \texttt{double} arguments and returns a \texttt{double} result (see also Section 3.9).

3.5.1 Type conversions and casts

From the table above, you can see that when only integer quantities are involved, integer arithmetic is used, with the effect that divisions can give unexpected results! This is often not what is desired and hence you should normally ensure that floating-point arithmetic is used in calculations, especially those involving divisions. If either \texttt{operand} of an operator is, say, a \texttt{double}, then the other operand will be ‘converted’ to \texttt{double} and evaluated. Otherwise, a good way of doing this is to specify whole number constants as \texttt{double}s by appending \texttt{.0} on the end of them; eg. 1.0, 0.0 etc.

The full rules that C uses to decide how and when to convert values from one type to another are fairly involved, but the most important ones are summarised below. In general, if an operation is being performed involving two quantities which have different types, the “smaller” type is promoted to the “bigger” type before the operation proceeds, and the result is that of the bigger type. This is the case whether the operation is arithmetic, or an assignment, or any other sort.

1. If either quantity is \texttt{double} convert the other to \texttt{double}.

\begin{verbatim}
  double answer, d = 2.0;
  int i = 4;
  answer = d/i;    /* The value of i gets converted to double
                   * because d is double */
\end{verbatim}

2. Otherwise, if either quantity is \texttt{float} convert the other to \texttt{float}.

\begin{verbatim}
  float answer, f = 2.0;
  int i = 4;
  answer = f+i;    /* The value of i is converted to float */
\end{verbatim}

3. Otherwise convert \texttt{char} to \texttt{int}.

\begin{verbatim}
  char c = 'a';
  int answer, i = 4;
  answer = c + i;
\end{verbatim}

In some cases the above rules may not be sufficient, however, you can if necessary force the value of any variable or expression to another type using a special operator called a \texttt{cast}. A cast is simply the name of the type you want the variable to be converted into (eg. \texttt{“int”} or \texttt{“double”}) surrounded by parentheses and placed in front of the thing you want to convert. eg. \texttt{“(int)floatvar”} converts the value in \texttt{floatvar} to an \texttt{int}. Casting does \textbf{not} permanently change the type of \texttt{floatvar} itself; it merely converts the type of its value for the current expression.

An example of when a cast is essential:

\begin{verbatim}
  int p,q;
  double d;

  p=1; q=2;
  d=p/q;    /* d will equal 0.0 because p and q are both ints */

  /* Now use a cast */
  d = (double)p / q; /* d will now equal 0.5 because the value of p
                    * (i.e. 1) was cast to a double forcing the
                    * division to be done with floating point
\end{verbatim}
Casts may seem odd, and perhaps a bit confusing, but they are actually very useful and straightforward. For instance, most of the mathematical functions in C (see Section 3.9) expect double arguments, and so you will often need to cast variables that you pass to them to double (some examples will be given later).

Casts are one example of how C is in general very permissive about what it allows you to do. Many languages are strongly-typed and are restrictive about which types they allow to be converted to which others. For experienced programmers, C’s flexible approach is extremely useful, but for beginners it can sometimes cause unexpected problems (the above being good examples).

Most languages will allow you to assign integer types to floating types, and to assign “smaller” types to “bigger” ones (eg. char to int or float to double). But another example of C’s permissiveness is the way that “bigger” types can also be assigned to “smaller” ones. For instance, C won’t stop you from assigning an 8-byte double to a 1-byte char (though it may give you a warning)! This can lead to problems if you’re not aware of what you’re doing, and so in general you should avoid doing so. If you do assign a floating point type to an integer type the fractional part will be discarded. Also, attempting to assign a number beyond the range of a smaller type will obviously cause problems and no guarantees can be made about what will then happen: if you do do this, you have again made a mistake and your program is faulty.

3.5.2 Some other operators

In addition to the usual arithmetic operators, assignments, and casts, C provides a range of useful and esoteric operators.

Increment and Decrement Operators

In particular, a group of operators that are widely seen in C programs are the increment and decrement operators, “++” and “--”. Very simply, the increment operator, ++, adds 1 to a variable, whilst the decrement operator, --, subtracts 1. Although they are nearly always used with integer types, it is perfectly legal to use them with floating-point types as well. They can be written either before or after the variable they’re operating upon, and this changes their behaviour subtly.

```c
int p, q;
p = 8;
q = p++; /* q is now 8; p is now 9 */
q = ++p; /* q and p are now both 10 */
```

So the difference between “p++” and “++p” is that p++ increments p but not until after its value has been used, whereas ++p increments p before its value is used. The decrement operator, --, works in exactly the same way. All forms of these operators can only be applied to variables, not constants (eg. “3++” is illegal). They can however be used simply to perform the increment or decrement operation, without the value being used. ie. “p++;” is a legal statement, and is effectively the same as doing “p=p+1;”.

Assignment Operators

Another group of operators which C introduced to the world are the compound assignment operators. An expression like “i=i+3” can be written instead as “i+=3”. There are a number of such assignment operators available, including one for each of the basic arithmetic operations:

```c
+= -= *= /= %=
```
These operators are not essential but they may produce more efficient code, and can even reduce the potential for mistakes. It should be stated that if there is an expression to the right of an assignment operator, then it is considered to be parenthesised; i.e.

\[
a *= z / x + 4;
\]

is equivalent to

\[
a = a * (z / x + 4);
\]

**Exercise 3.5**

Write a program which asks the user for a value for the radius of a circle (use printf and scanf) and calculates the area and circumference of the circle and then prints them together with the radius to the screen. You can use an approximate value for π. Don’t forget comments, and make sure the program’s output is descriptive, i.e. the person running the program is not just left with two numbers but is given some explanatory text.

### 3.6 Expressions, statements, and compound statements

So far we have concentrated on how C stores, manipulates and outputs data. These are all essential aspects of programming. Equally important, however, is the ability to be able to specify the order in which computations are performed. The programming jargon for this aspect of a language is *control flow*. We have already encountered the statement; C’s most fundamental control flow feature.

Formally, an *expression* such as “\[ x=0 \]”, or “\[ printf() \]”, or “\[ a+1 \]” becomes a *statement* when it is followed by a semi-colon. Because of this, in C the semi-colon is known as the *statement terminator*.

/* Here are two statements */

\[
x = 4;
x = x + 2;
\]

A group of statements surrounded by braces \{ \} is known as a *block* or *compound statement*. A compound statement is legal in C anywhere that a single statement can be used. Re-read that last sentence, it is very significant. As we shall see in the next few sections, it is also extremely useful.

/* Here is a single compound statement (or block) which contains 2 statements. It’s more readable if you indent. */

\[
\begin{align*}
x & = 4; \\
x & = x + 2;
\end{align*}
\]

One important note about the legal equivalence of single statements and blocks: the body of a function must always be a compound statement even if the function only contains one statement. For example it would not have been legal to have written our first *Hello world* program as follows:

/* THIS IS NOT A VALID C PROGRAM! */

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

So, if in doubt, use a compound statement, because they are always valid.

---

9This distinction between expressions and statements is a subtle one which many other languages don’t make. It’s helpful for being able to define things formally, but is not something you need worry about too much.
### 3.7 if statements

It is often essential in programming to be able to control the flow of a program according to whether some condition is true or false. The C construct for doing this is the *if* statement. It allows a statement to be executed conditionally. Its form is:

```
if(condition)
    statement
```

Where *statement* is only executed if *condition* is true. Note that an *if* statement only has a single conditional statement. However, as described above, you can arrange for a block of statements to be executed if the *condition* is true by using a compound statement.

There is an extended form of the *if* statement which allows alternative statements to be executed:

```
if(condition)
    statement1
else
    statement2
```

In this case, *statement1* is executed if *condition* is true whereas *statement2* is executed if *condition* is false.

What do “true” and “false” mean in this context? Computers, and their programming languages, depend heavily on a simple mathematical system known as Boolean logic. Boolean logic is based on a two-value, or binary, system of truth and falsehood, usually represented by 1 and 0.

C allows an expression of any type to be used as a condition in statements such as *if*, including *int* and *double*. In such cases, 0 is taken to mean false, and any non-zero value means true. The condition is almost always in practice a relational expression, i.e. a comparison of the relative values of two quantities. For example:

```
int a=2, b=3;
if(a < b)
    {
    printf("a is less than b\n");
    a=a+2;
    }
else
    {
    printf("a is not less than b\n");
    a=a-1;
    }
```

#### 3.7.1 Relational operators

C has a set of relational operators which take two arguments and yield an *int* result: 0 if the comparison is false and 1 if it is true.

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>is less than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>is less than or equal to</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>is greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td><code>==</code></td>
<td>is equal to</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>does not equal</td>
</tr>
</tbody>
</table>

The “ `==` ” is not a mistake: a single “ `=` ” is used for *assignment*, which is completely different to testing for equality, so a different symbol must be used. This is a major source of confusion and error in C! We’ve already

---

10To be explicit about Boolean conditions, you could include the header `<stdbool.h>` which defines a *bool* type and the constants *true* and *false*.  

---

Chapter 3. The elements of C
said that the condition expression can be any valid C expression, and that includes an assignment. Hence if you do
the following you will get very unexpected results:

    int a = 4;
    if (a = 5)
        /* ALMOST CERTAINLY A MISTAKE IT SHOULD BE: if (a == 5) */
        printf("a is 5\n");

In this code fragment, 4 is assigned to a and then it is intended to test whether a equals 5 and if so, output
an appropriate message. Unfortunately, the condition expression of the if is another assignment (not an equality
test) and so 5 is assigned to a and the result of that operation is also 5 (because in C, every expression has a
numeric value, and the value of an assignment operation is the quantity assigned). 5 also happens to be non-zero
and hence the condition is considered to be true and the printf() statement is executed.

It is almost inevitable that you will make this mistake at first, and it can be a very difficult one to spot because
it is perfectly valid C code (and can cause serious bugs in some situations). You will greatly reduce your chances
of making this mistake if you get into the habit early on of reading to yourself things like “a = 5” as “a becomes
5” or “5 assigned to a”; and “a == 5” as “a is equal to 5”. The compiler may emit a warning such as “suggest
parentheses around assignment used as truth value” which is an indication that a single = was used where two may
have been the intention. Some people like to write comparisons as “5 == a”, as the simple mistake “code5 = a”
leads to a compiler error (you cannot assign ‘5’ a new value).

3.7.2 Logical operators and else-ifs

Several if-else statements can be concatenated together to form an if-else-if statement

    if(condition1)
      statement1
    else if(condition2)
      statement2
    else
      statement3

Another important requirement is the ability to combine several conditions in one expression. For instance to
be able to do something if one thing is true AND simultaneously another thing is true. There are three logical
operators for doing this

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND: only TRUE if both operands are TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT: inverts its sole operand so that:</td>
</tr>
<tr>
<td></td>
<td>TRUE becomes FALSE; FALSE becomes TRUE</td>
</tr>
</tbody>
</table>

For example:

    /* Example of logically combining many conditions */
    if (a>b && a>c)
      biggest = a;
    else if (b>a && b>c)
      biggest = b;
    else if (!(c==a || c==b))
        /* ie. The opposite of "c equals a or b" */
        biggest = c;
    printf("The biggest is %d\n", biggest);

Relational operators have a higher precedence than logical operators; however, if ever in doubt, or to make
things clearer you may parenthesis (eg. “(a==b) || (a!=0)”).
The above example also illustrates that an else can contain another if statement (as indeed can an if statement). This is important because it allows a series of conditional tests to be chained together, although it can be the cause of confusing code. It also raises the issue of “to which if does an else belong?” eg. in the following example:

```c
if (a!=0)
  if (b==c)
    printf("a is non-zero and b equals c\n");
  else
    printf("err....which if do I belong to?\n");
```

The indentation suggests the right answer: the rule is that an else belongs to the first if above that hasn’t already got an else. If we want to force an else to belong to a different if we can do so using braces thus:

```c
if (a!=0)
{
  if (b==c)
    printf("a is non-zero and b equals c\n");
} else
  printf("Now I know!\n");
```

**Exercise 3.7**
(a) Write a program to read two numbers and print them in numerical order. 
*Consult a demonstrator before attempting part (b)*
(b) Write a program to read three numbers and print them in increasing order. It should be able to correctly cope with: (2, 6, 1); (1, 6, 2); (6, 1, 2); (4, 6, 4); (6, 4, 4); (4, 4, 6); and even (4, 4, 4)!

### 3.8 Loops and Iteration

There are many situations where it is necessary to be able to repeatedly execute a statement or a series of statements (for example when performing an iterative calculation). This control flow feature is known as a loop. Strictly, in C, only one statement can be used in a loop; however, as with if statements, if you need several statements to be repeated you simply have to group them together into a block.

There are three types of loop in C, for loops, while loops and do...while loops (the latter are used less often than the other two). We will look at each one in turn.

#### 3.8.1 for loops

When a given number of repetitions are required (for instance if the number of iterations needed for a calculation is already known) it is easiest to control a loop by using a variable as a counter. Three things need to occur in such loops: a counter must be initialised to a starting value before the loop is entered; its value must be checked every time around the loop (including prior to the first time) to see when to finish; and the value of the counter must be updated at the end of each loop iteration. In C this type of loop is implemented using a for statement.

The general format of a for loop is

```c
for(initialise ; condition ; update)
statement
```

The initialise part is an expression which is only done once before the start of the loop. It is virtually always an assignment used to initialise a loop counting variable (e.g. i=0).

The condition is an expression (nearly always a comparison to check the value of the loop variable) which is evaluated before the start of each iteration of the loop: if it is non-zero the statement is executed (e.g. i<10).

---

11 An example where whitespace assists the human, even though ignored by the computer.
After the statement has been executed, the update expression is evaluated. This usually increments the value of the loop variable (eg. \( i = i + 1 \), or \( i += 1 \), or most often \( i++ \)), though any expression is valid. The sequence then restarts at the condition. If the condition is false (zero), the loop ends.

Some examples will make this all a lot clearer. Suppose we want a program to print out the squares and the sum of the squares of the integers in the range 1 to 10. We could write:

```c
printf("%d	%d	%d\n", 1, 1*1, 1*1);
printf("%d	%d	%d\n", 2, 2*2, 1*1+2*2);
printf("%d	%d	%d\n", 3, 3*3, 1*1+2*2+3*3); etc.
```

but this is a particularly bad way of doing this. Imagine we wanted to change our program to print out the squares and the sum of the squares from 1 to 100! It is much easier to write this in a for loop as:

```c
int i;
int sumsq=0; /* Always set summation variables to 0 before use */
for(i=1; i<=10; i++)
{
    sumsq = sumsq + i*i;
    printf("%d	%d	%d\n", i, i*i, sumsq);
}
```

Here we set \( i \) as 1 initially. The first time through the loop \( \text{sumsq} \) will have the value \( 0 + 1*1 = 1 \), and this is printed to the screen. Next the variable \( i \) is incremented, and the condition \( i <= 10 \) is evaluated. It is true (\( i = 2 \)), so the loop is executed again. The second time through the loop \( \text{sumsq} \) is \( 1 + 2*2 = 5 \) and this is again printed to the screen. Now there is only one small change required to print the squares and their sums up to 100; change the condition in the for loop to \( i <= 100 \).

We’ll discuss one more example before looking at while loops. Suppose we want to print a conversion table between degrees and radians, counting down from 360 to 0 degrees in steps of 5 degrees. Before looking at the code fragment below, see if you can anticipate the initial value, test condition and update expression that will be required.

```c
int degree;
double radian;
for(degree=360; degree>=0; degree=degree-5)
{
    radian = degree * 2.0 * 3.1415926 / 360.0;
    printf("%3d	%.4f\n", degree, radian);
}
```

Notice that we can count backwards in a for loop because, as mentioned above, the update expression doesn’t have to be an increment.

### Exercise 3.8.1
(a) Write a program to calculate and output the sum of \( k^{-2} \) for \( k \) from 1 to 10 in increments of 1. Remember that integer division can give undesired results.
(b) More confident students may like to modify their part (a) program to calculate the same sum for \( k \) from 1 to \( N \) (where \( N \) should itself be varied from 5 to 50 in steps of 5). This will require you to use a loop within a loop (known as nesting). Think carefully about how to output the results neatly.

### 3.8.2 while loops

Sometimes, rather than executing a loop a fixed number of times it is necessary to continue executing it while a condition is true. This is done with a while loop which has a very simple and obvious structure:
The `condition` is tested first and the loop is only entered if it is true (non-zero). It is then re-tested each time around at the start of the loop. Once again, `statement` can be a compound block.

```c
int ok=1;
while(ok == 1)
{
    /* Do some stuff ... */
    if(the_end == nigh)
        ok=0;
}
```

Because the `condition` expression is checked before the loop is entered, it is worth noting:
1. that any variables (such as `ok`) which are used in the `condition` must already contain a value; and
2. that it is possible that the loop statement will never be executed.

It is also worth pointing out that if nothing in the loop affects the value of the conditional expression the loop will probably continue for ever! This is usually a bug. If you find that your program is not stopping this is more than likely the cause.

A useful example is to rewrite our example from the section on for loops as a while loop.

```c
int sumsq=0; /* Always set summation variables to 0 before use */
int i=1;
while (i<=10)
{
    sumsq = sumsq + i*i;
    printf("%d	%d	%d
", i, i*i, sumsq);
    i++;
}
```

Notice that the initial condition (`i=1`), the condition (`i<=10`) and the increment (`i++`) are still present. In this case, where the number of iterations is known before we enter the loop, a `for` loop is more compact and should be used.

### 3.8.3 do...while loops

There is a variation on `while` called `do...while`. It is not so widely used in C. It exists because it is occasionally useful to be able to guarantee at least one execution of the statement in a conditional loop.

```c
do
{
    /* perform task */
}
while(condition);
```

Note the semi-colon at the end: it is not optional. The difference here is that the `statement` is executed before `condition` is evaluated, thus guaranteeing at least one iteration of the loop.

One case where the `do...while` loop is useful is when you want to repeat a calculation until a user indicates via keyboard input that the program should stop.

The following piece of code will perform some calculation then ask the user if they want to repeat the calculation. If the user types `n` or `N` the loop will exit.

```c
char cont;
do
{
    /* perform task */

```
printf("Do you want to repeat the calculation? (y/n) ");
scanf(" %c", &cont); /* Note the space before %c*/

} while (cont != 'n' && cont != 'N');

Exercise 3.8.2
Find the largest factor of a number, n: test your program with different values of n. [Remember: a number’s largest factor is the largest integer that divides into the number with no remainder]

3.9  Mathematical functions

To keep the language small, the designers of C decided to build in only the most essential features, leaving everything else to be provided in one of a series of function libraries. We’ve already encountered exit() from the standard library, and of course printf() and scanf() from the standard input/output library. The advantage of this approach is that a rich and wide collection of C library functions have been developed over the years, and mathematical functions are no exception.

To gain access to most of the mathematical functions, you must include the following header file in your program:

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

The only exception is the abs() function which is an integer function supplied by the standard library, stdlib.h.

The most commonly-used maths functions are as follows (angles for the trigonometric functions must be in radians; 1° = 2π/360 radians):

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(x)</td>
<td>sine of x</td>
</tr>
<tr>
<td>cos(x)</td>
<td>cosine of x</td>
</tr>
<tr>
<td>tan(x)</td>
<td>tangent of x</td>
</tr>
<tr>
<td>asin(x)</td>
<td>arcsin of x</td>
</tr>
<tr>
<td>acos(x)</td>
<td>arccos of x</td>
</tr>
<tr>
<td>atan(x)</td>
<td>arctan of x</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>√x, x&gt;=0</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>x^y</td>
</tr>
<tr>
<td>exp(x)</td>
<td>e^x</td>
</tr>
<tr>
<td>log(x)</td>
<td>natural log, ln(x), x&gt;0</td>
</tr>
<tr>
<td>log10(x)</td>
<td>base 10 log(x), x&gt;0</td>
</tr>
<tr>
<td>erf(x)</td>
<td>error function of x</td>
</tr>
<tr>
<td>fmod(x, y)</td>
<td>floating point remainder of x/y with same sign as x</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>smallest integer not less than x (result is still double)</td>
</tr>
<tr>
<td>floor(x)</td>
<td>largest integer not greater than x (result is still double)</td>
</tr>
<tr>
<td>nearbyint(x)</td>
<td>nearest integer to x (result is still double)</td>
</tr>
<tr>
<td>fabs(x)</td>
<td>absolute value (ie. modulus) of x</td>
</tr>
<tr>
<td>abs(i)</td>
<td>as above for an int i</td>
</tr>
</tbody>
</table>

The math.h functions all expect double arguments and return a double value hence remember to use (double) casts for the arguments (as in the first example). Once again abs is the exception.

Notice that the arguments to some of the maths functions must be greater than zero. In particular, on this system if you try to pass a negative argument to sqrt() you will see the string NaN when you try to print the return value. NaN means Not a Number. Similarly, attempting to do things like log(0.0) or log10(-1.0) will yield -Inf (Minus Infinity), whilst attempting to divide by zero will of course give you Inf.

```c
#include <math.h>  /* NB. "math.h" not "maths.h"! */
```
```c
int p, q;
p=2; q=4;

/* We must cast both arguments passed to pow()*/
d = pow((double)p, (double)q);

/*.....Another example....*/
#include <math.h>
double x=16.0, s, cuberoot;
s = sin(x);
cuberoot = pow(x, 1.0/3.0); /*Note we’ve used 1.0/3.0 because 1/3 is */
/*an int expression (in fact equal to 0)*/
```

The `math.h` header file also defines several constants, including “M_PI” for \(\pi\), and “M_E” for \(e\) (i.e. the base of natural logarithms). Use these instead of generating your own \(\pi\) and \(e\), to reduce errors and ensure consistent results.

Be aware that the `floor()` function returns surprising results for negative arguments, e.g. `floor(-3.01)` returns the value \(-4.0\). The kind of rounding you will have been taught at school is implemented by `nearbyint()`.

**Exercise 3.9**
(a) Output a table of \(x\), \(\sin(x)\), \(\cos(x)\), \(\tan(x)\), and \(\sec(x)\) for 0 - 360 degrees in steps of 5 degrees. Check that your results are sensible.
(b) Find out what happens if you try to divide by zero, or take the log or square root of a negative number. The results may vary on different computer systems, and even with different compilers.
(c) Compute the cube root of a number to an accuracy of \(10^{-6}\) given that if \(a\) is an approximate cube root of \(x\) then \((2a + xa^{-2})/3\) is a better one.

### 3.10 Arrays

If a variable is a data structure that can hold one piece of data of a particular type (a single box), then an array is a structure that can hold several pieces of data all of the same type. Continuing with the box analogy (from Section 3.2) an array is a large box (or perhaps a rack) into which a series of smaller boxes can be put (figure 3.1). The smaller boxes, known as elements of the array, are numbered sequentially from zero so that they can be identified. The number, known as the array index, starts at zero, and goes up in increments of one.

```
xx
```

Figure 3.1: Arrays can be thought of as boxes in boxes

Arrays have types in exactly the same way as ordinary variables, such as `int`, `double`, or `char`. The individual elements of an array of `ints` behave exactly like ordinary `int` variables; they can store the same kind of data and can be used in exactly the same ways. In other words, the small boxes inside, say, an `int` “array box” are the same size and shape as individual `int` variable boxes.

In addition to providing an easy way of automatically creating lots of variables of the same type, arrays are especially useful because they allow their elements to be accessed by a numerical index.
3.10.1 Declaration

Arrays are declared in almost the same way as simple variables (and indeed declarations of arrays and variables can be mixed together): The declaration consists of the name of the type followed by the name of the array with a number in square brackets: the number specifies the number of elements the array will have. This number must always be an integer, and should usually be a constant; in many cases it is also possible to use an integer expression as long as the expression has a known value (e.g. if it contains a variable, that variable has been initialised) and is small; where small means $\sim 10^3$. Using a variable in an array declaration makes it harder for the reader to know how big your array will be, so consider restricting yourself to constant array sizes.

```c
char cc[10], c; /* Declares an array of 10 chars called cc, *
             * and an ordinary char variable, c */
int xx[4];
int i, j=3, readings[200];
double y[25], sum, voltage[j]; //voltage has three elements
```

3.10.2 Array elements

Since array indices start at 0, an array declared as "array[N]" will have elements called array[0], ..., array[N-1]. Once we have declared an array we can use its individual elements like ordinary variables of the same type. Array elements are referred to by the name of the array followed by their index number in square brackets; eg.

```c
y[2] = 6.5;
i = i * y[2];
```

However, normally arrays are processed en-masse, usually inside loops. For instance:

```c
sum=0.0;
for (i=0; i<4; i++)
    sum += xx[i]*yy[i];
```

3.10.3 Initialisation

C provides a way to initialise an array (ie. to assign initial values to all the elements) when it is declared. The syntax is as follows:

```c
int a[5] = { 5, 10, 15, 20, 25 };
/* The above is equivalent to */
int a[5];
```

In fact, if you are using this initialisation method, you may omit specifying the size of the array: the compiler will calculate the size to use by counting the number of initialisation values.

```c
int a[] = {5, 10, 15, 20, 25, 30, 35 }; /* Compiler will automatically
know to create int a[7] */
```

This is useful if you need to keep changing the amount of initial data you’re putting into an array but don’t want to have to keep matching the size of the array to the amount of data. On the other hand, it makes it harder for someone reading your code to know the size that the array will be, so be explicit with the size if you can.
Exercise 3.10

(a) Using a for loop, construct two 100 element arrays, \( x \) and \( y \), such that element \( i \) of \( x \) stores the value \( \sin(2\pi i/100) \) and the corresponding element of \( y \) stores \( \cos((2\pi i)/100) \). Print the values stored in the elements of \( x \) and \( y \) as you calculate them.

Consult a demonstrator before attempting part (b).

(b) Compute the scalar (i.e. dot) products \( x.x \), \( y.y \), and \( x.y \), to check that \( \sin \) and \( \cos \) are orthogonal. The scalar, or dot, product is defined as:

\[
x \cdot y = \sum_i x_i y_i
\]

3.11 Functions

*Functions* are the constructs C uses to separate logically distinct portions of a program into self-contained units. You’ve already encountered the `main()` function and have used many library functions: this section explains how to create your own functions.

There are several reasons why you might create a function.

1. A common one is to encapsulate a group of statements which might need to be executed more than once in different parts of a program (e.g. perhaps you need to be able to perform a particular task every time a certain situation arises): rather than having to type in the same group of statements multiple times, you can put them into a function, and then every time they need to be used, you simply call the function.

2. Another reason could be that your program comprises a series of logically distinct parts: rather than writing it as one long series of statements, it can be clearer to divide the code into sections, using a different function for each one. These can then be called in sequence.

3. A third reason could be that you are writing a portion of code that you are going to want to use again in other programs: if written properly, it is easy to copy a function into a new program and use it straight away.

In all of these scenarios, functions are being used to structure a program into *sub-programs*; i.e. modular sub-sections of the main program.

The form of a function definition is:

```
return-type function_name(argument declarations)
{
    declarations and statements
}
```

We can compare this with a function definition that we’ve already encountered, i.e. `main()`:

```c
int main()
{
    //do whatever your program needs
    exit(0);
}
```

Here we can see that the `main()` function is of type integer, and requires no arguments.\(^{12}\) `main()` is just like any other function except in one important respect; it is a mandatory function which will be run when your program is launched, or executed.

The arguments of a function are also sometimes referred to as its *parameters*. The *formal parameters* are the arguments used in the definition of the function, whilst the *actual parameters* are the arguments actually passed to the function when it is called.

Example of a function:

\(^{12}\)There are optional arguments to `main()`, explained in section 4.11.4.
int addup(int x, int y)
{
    int sum;
    sum = x + y;
    return sum;
}

Most functions return a value: as you can see from the example, this is done with a return statement. A function should always have at least one return, but may have several (this can be useful if you want to be able to pass back different values according to circumstances). In principle the main() function ought to have a return statement; but because we call the library function exit() within main(), it never gets a chance to return. Returning an exit status from the main function, such as return 0;, would be equivalent to quitting by using exit(0). The value passed back by return should be of the same type as the function (ie. int in the example above). It need not be the value of a variable, however; it could be a constant or even an expression. For instance, we could have written addup() as one line: {return x+y;).

Functions may return any of the C data types (eg. int, double, etc). By default, functions are assumed to return ints, but you should get into the habit of always explicitly declaring what type a function returns (even if it is int). Functions may even return no actual value at all. To get around the fact that, like variables, all functions must have a type, C uses a special type, called "void", for functions with no return value. You can return from void functions by using the statement return; with no parameter. Equivalently a void function will return when it “falls off the end”; that is when the closing brace at the end of the function is reached.

Once a function has been defined (or declared; see below) it can be called from another part of the program. If the function was defined with arguments then the number and type of arguments in the function call must match the arguments in the function definition.

An example of a program containing a couple of functions:

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

double my_function(double x)
{
    double answer;
    answer = exp(-x) / (x*x);
    return (answer);
}

void print_it_out(int a, double b)
{
    printf("First number is: %d\nSecond number is: %.4g\n", a, b);
    return;
}

int main()
{
    int i;
    double a, z;
    for (i=1; i<=10; i++)
    {
        a = i/10.0; /* Use 10.0 to force floating-point division */
        z = my_function(a);
        print_it_out(i, z);
    }
    exit(0);
}
```

3.11. Functions
Note that the variable \( a \) in \texttt{main()} which is a \texttt{double} is completely separate from the integer parameter \( a \) in the function \texttt{print_it_out()}.

In the example we have used two double variables in \texttt{main()}, \( a \) and \( z \), to store values that are then passed to the functions. We could have dispensed with these variables by using the expressions themselves as arguments. In other words, the second argument to \texttt{print_it_out()} could have been the call of \texttt{my_function()}, which in turn could have had \( i/10.0 \) as its argument. ie.

\[
\text{print_it_out}(i, \text{my_function}(i/10.0));
\]

This is perfectly legal because \texttt{my_function()} will be called and return a \texttt{double} value, which will then be passed as the second parameter to \texttt{print_it_out()}.

The functions have been \textit{defined} before \texttt{main()}. This is a good way of writing the code because before a function can be called, it must already have been declared or defined. A function declaration is the same as the definition but without the body of the function. ie.

\[
\text{double my_function(double x);} \\
\text{void print_it_out(int a, double b);}
\]

Note that in a declaration you don’t have to name the formal parameters, merely specify their types\(^\text{13}\). It would have been legal to declare both functions before \texttt{main()} and then put their definitions after \texttt{main()}: it’s up to you whether you declare first and then define later, or just define first.

Type the program in and run it for yourself to see what it does.

\begin{mdframed}
\begin{minipage}{.5\textwidth}
\textbf{Exercise 3.11}

The following function computes \( e^x \) by summing the Taylor series expansion to \( n \) terms. Write a program to print a table of \( e^x \) using both this function and the \texttt{exp()} function from the \texttt{math.h} library, for \( x = 0 \) to \( 1 \) in steps of \( 0.1 \). The program should ask the user what value of \( n \) to use.

\begin{verbatim}
double taylor(double x, int n)
{
    int i;
    double sum = 1.0;
    double term = 1.0;
    for (i=1; i<=n; i++)
        { /*Or we could have written: */
            term = term * x / i; /* term *= x / i; */
            sum = sum + term; /* sum += term; */
        }
    return sum;
}
\end{verbatim}

\end{minipage}
\end{mdframed}

\section{3.12 File I/O}

C communicates with files using a special type of variable called a \textit{file pointer}, written \texttt{FILE *}. Don’t worry about the syntax which looks worse than it actually is. Just follow the simple recipes given in this chapter for reading from and writing to files and you can ignore the details of what is going on and why. More information is given in section 4.11.

A file pointer variable, called \texttt{fout}, is declared like this:

\begin{verbatim}
FILE *fout;
\end{verbatim}

\(^{13}\)Such declarations are sometimes known as \textit{function prototypes}.
A file pointer is just like any other variable in C, you declare it at the top of your program and then assign a value to it. File pointers are associated with an actual file by using the `fopen()` function.

```c
FILE *fopen(char *filename, char *access mode)
```

(char * is one way of denoting a string). If the file cannot be opened `fopen()` will return a special value called `NULL`. You should check for this when you open a file otherwise you may try to write to a file using a pointer that doesn’t refer to the file which will give you errors. An easy way of doing this is

```c
FILE *fin;
fin = fopen("input.data", "r");
if (fin == NULL)
{
    /* oh dear, we can’t access the file...*/
    /* do something suitable here */
    printf("Cannot open %s\n", "input.data");
    exit(-1);     /*It’s nearly always necessary to*/
} /*exit if you fail to open a file*/

/* Note that a more compact way of opening
   * and testing the file pointer above is */
if ((fin = fopen("input.data", "r")) == NULL)
```

It may not look the easiest method of opening a file but if you use this method you will rarely go wrong. The access mode is a way of telling the compiler what we want to use the file for (e.g. reading or writing). The commonest access modes are:

- r  | Open a file for reading
- w  | Open a file for writing, if file already exists destroy current contents
- a  | Open a file for appending (i.e. adding to the end of the file)

If files are still open when a program exits, the system will close them for you. However it is good practice to close files explicitly when you’ve finished with them (it also means you can then open another file using the same file pointer). We can do this using the `fclose()` function.

```c
fclose(file pointer)
```

The table above lists modes for either reading from or writing to a file, but not both. There are modes which allow both reading and writing, but they are not straightforward to use so aren’t covered here. If you need to do both then write your output to the file, then close it before opening again for reading.

Let’s look at a proper example.

```c
#include<stdlib.h>
#include<stdio.h>
int main()
{
    FILE *fin, *fout;     /* Declare file pointers*/
    double x, y;         /* Other variables */
    if ((fin = fopen("input.data", "r")) == NULL)
    {
        printf("Cannot open %s\n", "input.data");
        exit(1);
    }
    if ((fout = fopen("output.data", "w")) == NULL)
    {
        printf("Cannot open %s\n", "output.data");
        exit(2);
    }
}
```

3.12. File I/O
fclose(fin);
fclose(fout);
exit(EXIT_SUCCESS);
}

3.12.1 Writing to Files

The example we have studied so far is not particularly interesting. We have only opened and closed two files. Writing to a file is not very difficult, you can do it with a modified version of the printf() function you have already seen. The function we use to write to files is fprintf(). fprintf() expects a file pointer as its first argument and then expects a format string similar to that for printf().

An example will make things clear.

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

int main()
{
  FILE *fout;
  int i;

  if ((fout = fopen("output.data", "w")) == NULL)
    {
    printf("Cannot open %s\n", "output.data");
    exit(EXIT_FAILURE);
    }

  for (i=0; i<10; i++)
    {
    fprintf(fout,"i = %d\n", i);
    }

  fclose(fout);
  exit(EXIT_SUCCESS);
}
```

Try typing this program in and run it. You should get a file called ”output.data” in your directory which contains the values 0...9. Note we are using fprintf() with a file pointer and not printf().

3.12.2 Reading from Files

We have already seen the fprintf() version of printf(). It will be little surprise to learn there is a fscanf() version of the scanf() function for formatted file input. Like fprintf(), fscanf() expects a file pointer as its first argument. The second argument is a format string similar to that for scanf().

EOF is a character which indicates the end of a file. It is returned by read commands when they try to read beyond the end of a file. It is possible to check for the EOF character using the feof() function which takes a file pointer and returns true if the file pointer is at the end of the file. A common way of reading to the end of a file is simply to use a while loop until feof() returns true.

```c
FILE *fp;
if ((fp = fopen("data.txt", "r")) == NULL)
    {
    printf("Cannot open %s\n", "data.txt");
    exit(EXIT_FAILURE);
    }
while (!feof(fp))
  {
```
/* read data from the file data.txt */
fclose(fp);

Expanding on the above example, read from the file data.txt which contains a column of ints and a columns of floats.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    int i;
    float x;
    if ((fp = fopen("data.txt", "r")) == NULL)
    {
        printf("Cannot open %s
", "data.txt");
        exit(EXIT_FAILURE);
    }
    while (!feof(fp))
    {
        fscanf(fp, "%d %f\n", &i,&x);
        /* Do something with i and x */
    }
    fclose(fp);
    exit(EXIT_SUCCESS);
}
```

Note, you MUST use \n in the format string of an fscanf() function to read over new-line characters when reading data from a file. This is different from the way scanf() is used to read data input from the keyboard, where \n should NOT be used.

**Exercise 3.12**
(a) Recreate the one hundred element arrays of sin and cos you created in Exercise 3.10. Print these arrays out to a file in the form of a table. The first line of the table should contain the first element of both arrays, the second line the second element, and so on.
Consult a demonstrator before attempting part (b).
(b) Create a program to read in the values from the file you have just created and print them to the screen.
This chapter contains information about various topics which do not belong in the previous chapter. Some may be needed later in the course, but are not directly connected with learning the basic language (such as the section on generating random numbers). Others are considered additional to the basic C course because they cover information about the language that is either of a reference nature, or is more obscure or advanced than that in the main part of the course.

If you have reached this point before the end of the second session, you should read through some or all of the sections in this chapter, and perhaps attempt the exercises. Otherwise, you should probably just skim through to familiarise yourself with the information included so you can refer back to any of it later if necessary. You should ask a demonstrator to assess you now for CO00 and assign and book one of the problems for you.

### 4.1 Random Numbers

In order to generate a random double in the range 0.0 to 1.0 you can use the \texttt{drand48()} function which takes no parameters, and returns a random number in the above interval. It is part of the standard library \texttt{stdlib.h}.

Ideally, before calling \texttt{drand48()} for the first time, you should call the function \texttt{srand48()} once to seed the random number generator. This does not return any value but does expect an integer seed value. The same seed value will produce the same sequence of random numbers (hence the tag “pseudo-random”!). (Although it is not recommended practice, constant default initializer values will be supplied automatically if \texttt{drand48()} is called without seeding the random number generator).

An example of using random numbers is

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

int main()
{
    FILE *fout;
    int i;
    double s, d;

    if ((fout = fopen("random.dat", "w")) == NULL)
    { printf("Cannot open random.dat\n"); exit(-1); }

    /* Seed the random number generator */
    srand48((unsigned int) time(NULL));

    /* get two random numbers */
    s = drand48();
    d = drand48();

    fprintf(fout, "%f\t%f\n", s, d);
}
fclose(fout);
exit(0);
```

\footnote{For those interested, \texttt{drand48()} generates pseudo-random numbers with a uniform distribution using the well-known linear congruential algorithm and 48-bit integer arithmetic.}
If you have time run the program above and plot the points in the file random.dat with gnuplot. You should see a random distribution in the graph. Use of gnuplot is explained on the Course's website at URL http://www-teaching.physics.ox.ac.uk/computing/UsefulSystemGuides/plotting.html; to know enough gnuplot just to draw the graph of random.dat, follow these instructions. Type gnuplot at a Terminal command prompt. This will start gnuplot and the prompt will change to gnuplot >; at this prompt type plot "random.dat". To quit from gnuplot simply type quit at the gnuplot prompt.

Optional Exercise 4.1
Write a program that simulates the UK National lottery by selecting six different whole numbers in the range 1 - 49.

The function drand48 is a Unix function and not available on Windows platforms. If you are compiling your program on a PC you should use the related but less capable rand function. This returns an integer between 0 and RAND_MAX, so if you want a double between 0 and 1, like drand48() creates, do:

double random_number;
random_number=rand()/(double)RAND_MAX;

4.2 Complex numbers

The latest standard version of the C language, C99, includes complex number types and a host of functions to deal with them. You must include the complex.h header to be able to use complex numbers; this defines the complex word for creating complex variables and the preprocessor constant I, defined as I=√−1.

Complex variables are of some floating–point precision, we shall just use double in these examples as it’s more precise than the float alternative. They can be declared and operated on much the same as any other variable:

double complex z1,z2,z3;
z1=3.0+4.0*I;
z2=-3.0+2.5*I;
z3=z1+z2;
z3*=(1.2-2.4*I);

here the similarity with regular, real variables ends. For instance it isn’t possible to print a double complex directly using printf(). Instead we have to pick out either the real and imaginary parts, or the magnitude and argument.

double complex z1=3.0+4.0*I;
printf("cartesian:\t%lf + %lf i\n",creal(z1),cimag(z1));
printf("polar:\t\t[ %lf, %lf ]\n",cabs(z1),carg(z1));

Other common functions performed on complex numbers include taking the complex conjugate using the conj() function; the square root with csqrt(); and the complex exponential:

double gamma,omega,V0,t; /* gamma is the decay constant
   * omega is the angular frequency
   * V0 is the initial voltage at time t=0
   */
double delta_t = 0.1;
double complex voltage;
// initialise the variables somehow
// ...
for (y=0; y<=2; y++)
{
for (x=0; x<=4, x++)
    something = V[y][x] * or_other;
}

It is natural to want to think in terms of (x,y), but in C you should always think (y,x) or “row, column”. Remember that the indices run from 0...N-1: ie. we had to declare V[3][5] but the last element of the array is V[2][4].

Finally, it is possible to have 3- and higher-dimensional arrays in C; for instance, the syntax for a 3-D declaration would be:

```c
int threeD[2][5][10]; /* This creates a 2x5x10 int array with
              * elements threeD[0][0][0] ....
              * threeD[1][4][9] */
```

As with 2-D, it is the right-most index which changes fastest, then the middle one, and the left one changes slowest. One could keep extending the principle to many higher dimensions. However, 3-D arrays are already unwieldy and in practice one rarely uses more than 2-D arrays, and even those are avoided when simple arrays can be used. The general rule, as always in programming, is to keep things as simple as possible.

### 4.4 String manipulation functions

A string is a sequence of zero or more characters surrounded in double quotes, for example

```
"I am a string"
"Hello World!"
"
/* The empty string */
```

The double quotes are not part of the string, they only serve to delimit it. The C programming language does not have a string data type, instead a string is stored as an array of characters with a NULL (’\0’) character at the end. When you need a character array to hold a string you need to declare the array to be one element larger than the string so that there is room for the NULL character. For example, if you are declaring an array called answer to hold the value “false” you should declare it as

```c
char answer[6];
```

Even though there is no string type in C, strings are so useful that there is an extensive set of library functions for manipulating strings.

To gain access to the string functions, you must include the following header file in your program:

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

The most commonly-used string manipulation functions are as follows:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcmp(s1, s2)</td>
<td>Compares s1 to s2, returns 0 if s1 == s2</td>
</tr>
<tr>
<td>strncmp(s1, s2, int n)</td>
<td>Compares the first n characters of s1 and s2</td>
</tr>
<tr>
<td>strlen(s1)</td>
<td>Get the length of the string s1</td>
</tr>
<tr>
<td>strcpy(s1, s2)</td>
<td>Copies string s2 into s1</td>
</tr>
<tr>
<td>strncpy(s1, s2, int n)</td>
<td>Copies n characters from string s2 into s1</td>
</tr>
<tr>
<td>strstr(s1, s2)</td>
<td>Concatenates string s1 onto the end of s1</td>
</tr>
<tr>
<td>strncat(s1, s2, int n)</td>
<td>Append n characters from string s2 to string s1</td>
</tr>
</tbody>
</table>

For example:
/ Example 1 */
#include <stdio.h>
#include <stdlib.h>
/*The next line must always be used with string functions*/
#include <string.h>

int main() {
    char reply[4];

    printf("Do you want to run this program? [yes/no] ");
    scanf("%s", reply);

    if(strcmp(reply, "no") == 0) /* 0 means they are the same */
        exit(1);
    else
        /* The rest of the program... */
}

/* Example 2 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char str[100];

    printf("Enter a string
");
    scanf("%s", str);
    printf("The length of (%s) is %d 
", str, strlen(str) );

    exit(0);
}

/* Example 3 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char firstname[20], surname[20];
    char name[42], name_official[42];

    strcpy(firstname, "Tom");
    strncpy(surname, "Jones", strlen("Jones");
    strcpy(name, firstname);
    strcpy(name_official, surname);
    strcat(name, surname);
    strcat(name_official, surname);

    printf("name = %s\n", name);
    printf("official name = %s\n", name_official);

    exit(0);
}

4.4. String manipulation functions
Optional Exercise 4.4
Write a function that returns true if an input string is a palindrome. A palindrome is a word that reads the same backwards as it does forwards e.g. ABBA.

Another very useful pair of string manipulation functions are `sprintf()` and `sscanf()` which are in the `<stdio.h>` library. They are similar to `printf()` and `scanf()` but instead of printing a formatted string to the screen (which we refer to as standard output or stdout) or reading a formatted string from the keyboard (standard input or stdin) they print and read formatted strings to/from strings.

For example:

```c
#include <stdio.h>

int main()
{
    char str[20];
    int i;

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

    sprintf(str, "You entered %d\n", i);
    printf("%s", str);

    exit(0);
}
```

4.5 switch statements

A switch statement allows a single variable to be compared with several possible integer constants. It can often be used instead of a sequence of the if-else-if statements we have seen before. The syntax for the switch statement is

```
switch ( variable )
{
    case integer_constant1:
        statements...;
    case integer_constant2:
        statements...;
    default:
        statements...;
}
```

Each case is checked in turn and if the variable matches the value of a case constant those statements are executed and then all the statements in the subsequent blocks are executed until execution reaches the end of the switch statement. It is a common mistake to think that execution of the switch statement ends once the statements matching the correct case are executed. For this reason a break statement is usually put at the end of each block of statements. An example will make this clear.

Note: the default case is a special one that is executed if no other constant in the switch statement matches the variable. The default case is optional.

For example, suppose we have an integer which is read, say, from the command line. Until now we would check it as a series of if-else-ifs.

```
if (q==1)
{
    printf("q equals 1\n");
}
```
else if (q==2)
{
    printf("q equals 2\n");
} else if (q==3)
{
    printf("q equals 3\n");
} else
{
    printf("q does not equal 1, 2 or 3\n");
}

However, using a switch statement can be a neater solution.

switch (q)
{
    case 1:
        printf("q equals 1\n");
        break;
    case 2:
        printf("q equals 2\n");
        break;
    case 3:
        printf("q equals 3\n");
        break;
    default:
        printf("q does not equal 1, 2 or 3\n");
}

The break statement is used to stop executing the switch statement at the end of each case. If you forget the break statement, execution will continue with the next case. Sometimes, though, it actually is useful to be able to execute a block of statements for a range of cases, e.g. if (letter=='A' || letter=='E' || letter=='I') etc if we are counting vowels. The next example leaves out the break statement to show you how to match cases with a switch statement.

```c
int numberofvowels=0, numberofspaces=0, numberofconsonants=0;
char letter;
/*Some code which inputs something into letter*/

switch (letter)
{
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U':
        numberofvowels++;
        break;
    case ' ':
        numberofspaces++;
        break;
    default:
        numberofconsonants++;
}
```

If letter contains 'E' then we start executing the first block of statements under the 'E' option. You’ll notice that there is no block under the 'E' case. Since there is no break statement execution continues with the 'I' case, and then the 'O' case and then the 'U' case where we increment numberofvowels. Since there is a break statement here we ‘break’ out of the switch statement.
The default option is a special option that is executed if no other option in the switch statement is executed. In our last example, if letter was ‘t’ then none of our conditions (‘A’, ‘E’, ‘I’, ‘O’, ‘U’, ‘’) are met so the statements after the default case are executed.

Remember: switch statements can ONLY be used with integer types (ie. int, char, short, long, etc) and that the values after the case’s must evaluate to integer constants.

4.6 The ternary operator

The C language contains only one ternary operator; i.e. one operator that requires three arguments. That is the ? operator which is used thus:

condition ? expression1 : expression2;

The invocation of the ternary operator is equivalent to, though more concise than, the following code:

if(condition)
  statement1;
else
  statement2;

The ternary operator really comes into its own thanks to the way in which C evaluates expressions. Most expressions\(^2\) can be evaluated to give some result. As an example, the expression \(x=3\) assigns the value 3 to the variable \(x\), and evaluates to 3. The ternary operator can then be used to further streamline code using this evaluation rule. For instance the fragment:

\[
\text{if } \left( \text{year mod 4} \neq 0 \right) \\
\text{leapYear} = 0; \\
\text{else} \\
\left\{ \\
\text{if } \left( \text{year mod 100} \neq 0 \right) \\
\text{leapYear} = 1; \\
\text{else} \\
\left\{ \\
\text{if } \left( \text{year mod 400} \neq 0 \right) \\
\text{leapYear} = 0; \\
\text{else} \\
\text{leapYear} = 1; \\
\right\}
\right\}
\]

may be reduced using the ternary operator to:

\[
\text{leapYear} = \left( \text{year mod 4} \neq 0 \right) \ ? 0 : \left( \text{year mod 100} \neq 0 \right) \ ? 1 : \left( \text{year mod 400} \neq 0 \right) \ ? 0 : 1;
\]

4.7 More about variables

4.7.1 Scope

The scope of a variable is the region of the program in which it is accessible. Variables declared inside functions (which are the only sort we have so far encountered) are accessible only from within the function in which they are declared. They are thus sometimes referred to as local variables.

This also means that it is possible to have local variables with the same name in different functions. In terms of the box analogy, you can have different boxes in different parts of the program with the same label. The contents of these different boxes are entirely separate and cannot be mixed up. For example:

\(^2\)The exceptions are functions of type void.
```c
#include <stdlib.h>
#include <stdio.h>

void afunction()
{
    int i;
    i = 4;
    printf("The value of i inside afunction() is: %d\n", i);
    return;
}

int main()
{
    int i;
    i = 1;
    printf("The value of i inside main() before is: %d\n", i);
    afunction();
    printf("The value of i inside function afunction() is: %d\n", i);
    exit(0);
}
```

Running the above program should generate the following output; as you can see, the value of `i` in `main()` is unaffected by the value of `i` in `afunction()`.

The value of i inside main() before is: 1
The value of i inside function afunction() is: 4
The value of i inside main() after is: 1

### 4.7.2 Global Variables

Up until now all the variables used have been local. C does allow variables which are external to any function and thus globally available to all functions. This is done quite simply by putting the variable declaration outside all your functions. In other respects, they look the same. Try entering and running this program

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

int a_global_variable;

void afunction()
{
    printf("%d\n", a_global_variable);
    return;
}

int main()
{
    a_global_variable = 5;
    afunction();
    exit(0);
}
```

Global variables are useful because they allow different functions direct access to the same data; however, they are also potentially dangerous because they allow bugs in one part of a program to affect the data used by a different part of the program. They also reduce the modularity of your code: for example, if you have written a particularly useful function you may want to re-use it in other programs\(^3\). However, if the function you want to

\(^3\)Such code re-use is a very good aim; it saves time and tends to make programs more reliable, because the bugs in re-used code are more likely to have already been found and fixed.
re-use depends on global variables it makes it much harder to use the function in a different program, because you also have to ensure that the same global variables are made available and that they have meaningful values. For both these reasons, it is best to avoid using global variables except when they make your program much clearer or more efficient. In general you should pass data between functions using arguments and return values.

Note that global variables are only in scope for functions which appear after the global variable has been declared. Hence, it is normal for global variable declarations to occur at the top of a program, usually after header file includes. Remember to initialise global variables before using them; calling a function() without giving a global variable a value would have unpredictable results.

4.7.3 Storage classes

We have seen that the scope of local variables is limited to the function they are declared in. The effect of this is that when the function returns, the variables are no longer used and are destroyed, we say the variable has gone 'out of scope'. What happens if you actually want to remember the value of a variable between function calls? One option is to use a global variable but as we have seen this is not always wise. Instead we can tell the compiler to remember the value of a variable in a function between calls to the function by using the keyword static before declaring the variable.

Compile and run the following piece of code as an example.

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

void counter()
{
    int i=0;
    static int j=0;
    printf("The value of i is: %d\n", i);
    printf("The value of j is: %d\n", j);
    j = j + 1;
    i = i + 1;
    return;
}

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

4.8 The C Preprocessor

As its name suggests, the C Preprocessor processes the code of your program before the compiler proper sees it. It is essentially a fairly simple utility which performs textual substitutions on your code according to directives you give it. Preprocessor directives are identified as lines which start with commands prefixed by the hash character, "#", and can occur at any point in your program. In practice, however, they are normally confined to the top of a program. There are several directives available, however two of them are far more widely used than any of the others: #include and #define.

You have already used #include on numerous occasions. What it literally does is to replace itself with the contents of the file specified. Normally it is used to include standard library header files, and header files written by the programmer. In principle, though, it could be used to include any file.

#define is largely used as a means of being able to define easily identifiable constants which may be used throughout a program, and which can be modified easily in one place. By convention, #define constants all have upper case names to help differentiate them from variables and other names used in a program. You
should definitely stick to this convention in your programs. The values of such constants can also be overridden at
compile–time which can be a convenient way of altering some default setting for a program without having to edit
it. In its simplest mode, it is used as follows:

```c
#define CONSTANT 1.0
#define NAME "Boris" /* They can be any text you like, not just
#define TRUE 1 numerical values or strings */
/* Then later in your program you might do things like */
y = CONSTANT * x + c;
if (y == TRUE)
    printf("Hello %s\n", NAME);
```

Note, however, that `#define`’d constants do not get substituted inside strings.

It is important not to put a semi-colon at the end of a `#define` directive because the value of the `#define`
is substituted straight into your code (semi-colon as well) which is generally not what you want. Remember the
rule ‘only put a semi-colon at the end of a statement’.

More complicated substitutions can be performed using `#define`, which is really a `macro`
environment. Constants such as those described above are just trivial examples of macros. Because preprocessor directives are
substituted before the program is compiled, such macros can be independent of variable type (where appropriate).
A bad example would be this:

```c
#define SQUARE(s) s*s
/* later on... */
x=a+b*y+x*SQUARE(y);
z=SQUARE(x+y);
```

Why is this bad? Remember how the preprocessor works; the contents of the macro are substituted `verbatim`
into the code, which is then compiled. So the second line of C in the above example would expand into this:

```c
z=x+y*x+y;
```

which due to the higher precedence of the multiplication operator (see section 3.5) is the same as \( z = xy+x+y \),
not \( z = (x+y)^2 \) which the programmer may have intended. A better macro would be:

```c
#define SQUARE(s) (s)*(s)
```

which would expand to \( (x+y)*(x+y) \), leading to fewer surprises when the program is run. Even this is not
robust enough to cover all uses; it is generally a good idea to avoid the increment and decrement operators (section
3.5.2) in macros or in code that calls a preprocessor macro.

## 4.9 Data structures

Let us say that we were writing a program to browse a catalogue of stars. We might have a list of facts which it
must contain for each star in the catalogue:

- **name** The name of the star (string)
- **spectral type** The star’s stellar spectral type (character)
- **distance** Distance from the Sun (integer light years)
The apparent magnitude (i.e. brightness, floating-point)

Each of these facts is called a field, and all of the fields for a given star comprise that star’s record. Note that the fields each have a different type. C has the data structure, or struct, which lets us build a variable type to represent these records.

```c
struct star
{
    char *name;
    char spectral_type;
    int distance;
    double magnitude;
};
```

You can now declare variables to be of type struct star just as you would declare integers of type int. The individual fields in any record, known as the struct’s members, can be accessed by following the record’s variable name by a ‘.’ and the member name. This is how we might declare a struct for the Sun and use it:

```c
struct star the_sun;
the_sun.name="Sol";
the_sun.spectral_type='G';
the_sun.distance=0; //the Sun isn’t far from the Sun at all!
the_sun.magnitude=-26.73;
printf("The Sun is a type %c star.\n",the_sun.spectral_type);
```

It is also possible to define a structure when it is declared, as long as the members are defined in the order they appear in the struct declaration.

```c
struct star proxima = {
    "Proxima Centauri", //name
    'M', //spectral type
    4, // distance to nearest light year
    11.05 // apparent magnitude
};
printf("%s has an apparent magnitude of %g\n",proxima.name,proxima.magnitude);
```

The members of a struct are not restricted to simple types – they could be arrays, other structures, or even a pointer to its own struct type!4

### 4.9.1 union

A union is closely related to a struct, and is manipulated via the same method with the keyword union replacing the keyword struct. The difference is that when a struct is declared, enough space is reserved to store each of its member variables; a union stores each of its members at the same memory location and is only as big as its largest member. It is therefore important in dealing with unions to keep track of which member you are trying to access, as the value may not make sense if you store e.g. a float then try to read it back as an int. When a union is initialised, the value with which it is initialised must be that of the union’s first member. So the union:

```c
union the_value
{
    int intval;
    float floatval;
    double doubleval;
    char *stringval;
};
```

4A struct could not have its own struct type as a member, as the type is not completely defined until the closing curly brace. A pointer is fine however, and this is a common way to define lists of records. See section 4.11 for more on pointers.
must be initialised with an integer.

It is very unlikely that you will use the union type in the computing course, and we will not describe its
use in any greater depth. The union is frequently found in operating system programming, especially network
programming where the format of data to be processed depends on context.

4.10  Enumerated types

Enumerated types, or enums are used to name (i.e. enumerate) the elements of a finite set, and you can declare
variables that can be represented by that set. This can greatly reduce the code required to define and work with
a long list of constants. For instance, we could use the preprocessor to assign a numerical value to each of the
months of the year:

```c
#define JAN 0
#define FEB 1
// ...
#define DEC 11
// ...
int the_month=OCT;
```

this could be replaced by an enumerated type:

```c
enum month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
// ...
enum month the_month=OCT;
```

Using an enumerated type ensures that the variable the_month can only take on values equivalent to JAN–
DEC, unlike the integer in the first code fragment which can have values that are nonsensical in this context. A
particularly important implication of this is that switch statements do not need a catch-all default: branch,
as all of the possible outcomes are known in advance.

4.11  Pointers

4.11.1 What is a pointer?

Although we have not explicitly discussed the use of pointers so far, they are such an integral part of the C language
that you have already been using them. When declaring a string as char * or declaring FILE * for accessing
a file, you are using pointers. When writing code such as:

```c
int i;
scanf("%d",&i);
```

you are using pointers. A pointer is the address in memory at which a variable is stored, and can therefore be
thought of as a reference to that variable. To declare a variable as a pointer to a particular type, put an asterisk (*)
before its variable name in the declaration. Asterisks are also used to get the value stored by the variable a pointer
references. To get a pointer to an existing variable, prefix its name with an ampersand (&). This needs further
explanation, so consider this code fragment:

```c
int a,*b,c;
/* a and c are integer variables
   * b is a pointer to an integer
*/
a=3;
b=&a; // b now ‘points to’ a
c=*b; // c is now the same variable as a
```

The &a construct is called referencing the variable, and *b is known as dereferencing the pointer. Uses for
these constructs, especially dereferencing, are described in the rest of this section.
4.11.2 Pointers and arrays

The type of a variable declared as an array is a pointer to the type in the array’s elements. So if there is an array
int xx[7], the variable xx is of type int *. This is why strings can be declared either as char s[] or as
char *. Knowing that an array variable is also a pointer is less useful than the reverse – it is possible to use a
pointer as a reference to an array, provided we know something about memory allocation.

It is not possible in C to declare an array with a variable size:

```c
/*** THIS IS INVALID C ***/
int n;
scanf("%d",&n);
double xx[n];
```

however, as we now know that a pointer and an array type are interchangeable, we can declare a double *
pointer and use that as an array. In order to make use of an array of n doubles, we have to allocate enough
space in memory to hold them (otherwise we may be trying to read and write into space used by something else,
which will cause nonsensical results or the program to fail). This can be done using the malloc() function,
which returns a pointer to an area of memory of the size specified in its argument. How do you know how big a
double is? The sizeof() function can tell you.

```c
int n;
double *xx;
scanf("%d",&n);
xx=malloc(n*sizeof(double));
/* xx now points to space equivalent to n doubles
* it can be treated as an array xx[0],xx[1],...,xx[n-1]
*/
```

Optional Exercise 4.11
Recreate the table of sines and cosines from exercise 3.10, but this time ask the user for the number
n of rows in the table.

4.11.3 Pointers and functions

The variables passed to a function call inside the brackets (e.g. x and y in pow(x,y) are called the function’s
arguments. The arguments to a function are immutable inside the function, for instance the pow(x,y) call cannot
modify either x or y. What do we do when we want a function to be able to change its arguments? You have
already seen the answer when using the scanf() function; if a function must modify the value of a variable, a
reference (i.e. pointer) to that variable is passed instead. Now it is the pointer that is immutable, but the pointer
can still be dereferenced and the variable modified.

Passing arguments by reference is so important in dealing with functions that take structs as arguments that
C has a special syntax to access a member of a struct through a pointer. Consider this function:

```c
void scalar_mult(struct vector *z,int n)
{
    // remember that *z will dereference the pointer
    (*z).x *= n;
    (*z).y *= n;
    return;
}
```

this can be rewritten in a less unwieldy manner:

```c
void scalar_mult(struct vector *z,int n)
{
    z->x *= n;
    z->y *= n;
}
```
4.11.4 Command line input

If you have used the Unix terminal described in section 2 at all, you will have noticed that many programs can take multiple arguments on the command line, such as `rm` which accepts a list of files to be deleted. These programs are all written in C, so it must be possible to make use of these command line arguments in any C program.

So far, all of the complete programs in this book’s examples have declared the `main` function as:

```c
int main()
```

In fact, another declaration is permitted:

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

But what’s a `char **`? It is a pointer to a pointer to a `char`, or as a pointer can also be an array and an array of `chars` is a string, `argv` is an array of strings. The number of elements in this array is `argc`, so it runs from `argv[0]` to `argv[argc-1]`. The string `argv[0]` is the name of the program and the remaining elements are the arguments that were passed on the command line. Here is an example of how to use `argc` and `argv` in a program which expects exactly one argument, that of a filename:

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

int main(int argc, char **argv)
{
    FILE *fp;
    // first check that we have the appropriate number of arguments
    if(argc!=2)
    {
        printf("Error: usage is '%s <filename>'\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // OK, try to open the file
    fp=fopen(argv[1],"w");
    /* do whatever we need to
     * such as check that the file opened successfully
     * ...
     */
}
```

4.12 Custom variable types with typedef

C allows us to define our own types using a `typedef` command.

The `typedef` command ‘defines’ a new variable’s type. The `typedef` keyword can be used to create any definition you like, for example to declare a new data type called `digits` (which we’ll make integers):

```c
typedef int digits;
```

which we could use to declare variables.

```c
digits a,b,c;
```

Now `a, b, c` are variables of type `digits` (which in turn are `int`s). This may not seem to be particularly useful, but is common in systems programming to make the `intent` of a variable more obvious. For instance, a number of the standard library functions to manipulate times take as an argument the number of seconds since a specific date. This could be expressed as an integer, but the type is redefined through `typedef` so that it’s called `time_t` in these functions.

It’s more common to see `typedef` used alongside more complicated data types, such as structures. A program modelling nuclear physics could define the following type:

```c
struct particle
```

[5]In fact, the shell may do some manipulation of them but will still pass a list of arguments into the `argv` array of the program.
typedef struct
{
    int neutrons;
    int protons;
} nucleus;

and subsequently use the nucleus type:

nucleus helium;
helium.neutrons=2;
helium.protons=2;

This typedef could be combined with another typedef for a pointer to the structure. As functions which act on structures commonly take a pointer as their argument, this is a frequently-used construction.

typedef nucleus *nucleusPtr;

/* this function works out the daughter of an alpha decay.
   * Whether the decay is actually possible is not considered.
   */
void alphaProduct(nucleusPtr parent)
{
    parent->neutrons -= 2;
    parent->protons -= 2;
}

4.13 Variable–argument functions

This section is very advanced and requires understanding of pointers, see section 4.11. Do not worry if this section is beyond you as you will not need to use the techniques here on the course, it is covered for completeness.

4.13.1 Introduction to variable–argument functions

All of the function declarations you have written so far define all of their arguments explicitly, and there is no flexibility in the type or number of arguments passed. But you have also used standard functions which can take a variable number of arguments, such as printf() and scanf(). It is possible to construct such variable-argument; or vararg, functions in your own code.

4.13.2 Declaring a variable–argument function

A good example of a varargs function is printf(), the declaration for which may look like this:6

```c
int printf(char *format,...);
```

The ellipsis (...) does not indicate that something has been left out, it is the syntax for showing that a variable argument list follows. The format argument is a fixed argument - all varargs functions must have at least one fixed argument and all fixed arguments must precede the ellipsis. The function must be able to determine for itself how many arguments are in the list and of what type they are, and this is usually encoded in the fixed arguments. In the case of printf() the format string contains the conversion codes (section 3.3) which encode the expected number and type of the arguments in the varargs list.

4.13.3 Accessing the varargs list

There is no way that a function can know how many arguments have been passed or of what type they are, when they are part of a varargs list. It must be able to find this out for itself. As described in section 4.13.2, one way

---

6It doesn’t look like this in reality, it depends on keywords which have not been covered here.
to do this is to encode it into the fixed arguments of the function. A more robust way, if dealing with arguments expected to be of exactly one type, is to require the last entry in the list a known value that couldn’t be mistaken for a real argument – e.g. NULL.

The list of arguments represented by the ellipsis are available through a special type called _va_list_. This list is initialised by the macro va_start(), and you can iterate through the argument list using va_arg(). Once you have finished using the argument list you must clean up the memory etc. allocated to it by calling va_end(). In this example, taken from the GNU C library manual, we create a function _add_em_up()_ which returns the sum of _n_ integers, passed to it in a variable argument list.

```c
#include<stdio.h> //header file defining the va_* macros

int add_em_up(int n,...) {
    va_list ap;
    int i, sum=0;
    va_start(ap,n); /*tells the list to start after the named
        *variable, n
    */
    for(i=0; i<n; i++)
    {
        sum += va_arg(ap,int); /*get the next argument from the list,
            *treating it as type integer
        */
    }
    va_end(ap); //clean up
    return sum;
}
```

Although the features here look like functions, they do things that no C function could possibly do and therefore must be implemented as compiler macros.

---

4.13. Variable–argument functions 63
Reserved Words and Program Filenames

The C language reserves special words, which we call keywords, you must not use any of C’s keywords as your variable names. The words reserved by C are:

auto   do   goto   return   typedef
break  double  if   short   union
case   else   inline  signed  unsigned
char   enum   int   sizeof  void
const  extern  long   static   volatile
continue  float   register   struct   while
default for   restrict   switch

do not use any of the following words either (although they are not C reserved words, they conflict with the names of commonly-used C library functions):

abort  clock  getenv   rand   srand
abs   close  labs  read   strcmp
atof   div   malloc  remove  strcmp
atoi   exit  open  rename  system
atol   free  printf  scanf  time
calloc  getchar  putchar  signal  write

You should also avoid all the names defined in the math.h library (you must avoid them if you #include <math.h>):

acos  cos  floor  sin  tanh
asin  cosh  log  sinh
atan  exp  log10  sqrt
ceil  fabs  pow  tan

The above two lists of standard library function names are far from exhaustive; however they do include the majority of commonly-used functions and/or functions with names that might conceivably be chosen unwittingly.
Creating Larger Projects

Programmers will often separate large projects into multiple files rather than keep all of the source in one large code file. The reasons for this include organisation; code re-use (a file containing a couple of generally useful functions can be included in more than one project) and efficiency (if a change is made to a small part of the project, only the affected files need be recompiled).

XCode is designed to support multiple files in a project; adding a new file to an existing project is done by selecting ‘New File...’ from the File menu in XCode. However, XCode is only available when you are using a Mac locally (i.e. are sat in front of it, rather than remotely connected) and therefore it is not necessarily available in all situations. In section B.2 the command-line tool make, which is available on nearly all Unix systems, is described. Using make it is easy to automate the compilation of projects that contain multiple source code files.

## B.1 Header Files

You will already be familiar with the preprocessor directive `#include`, which is replaced with the contents of the specified file before the compiler takes any action. So far you have been using the directive to refer to header files that are built into the system e.g. `#include <stdio.h>` but it is also possible to create your own header files; files containing C source code but with the `.h` extension.

Header files are usually used when a C program file contains a number of functions; the function declarations can be kept in the header file while the bodies of the functions – their definitions – go into the associated `.c` file. This separates the code’s interface (what it can do) from its implementation (how it does it); a header file containing appropriate comments can act as a perfectly adequate form of documentation for your functions.

As an example of how the interface and implementation are split when header files are used, consider this header file `saysomething.h`:

```c
/* saysomething; functions to print out polite messages */

/* say_hello() prints a greeting message */
void say_hello(void);

/* say_goodbye() prints a departing message */
void say_goodbye(void);
```

The associated C file, `saysomething.c` may look like this:

```c
#include "saysomething.h"

void say_hello(void)
{
    printf("Well, hello there!
"");
}

void say_goodbye(void)
{
    printf("Goodbye, take care!
"");
}
```

Note that the `#include` in the C file uses quote marks instead of angle brackets. This tells the preprocessor to look for the file in the current directory instead of searching the system include paths. Also note that you – and any other programmers who have to look at your code – can tell something about what the two functions should do without having to inspect the full source code. This would be invaluable if the functions were much longer than the simple examples described here.
B.2 Makefiles

This section is only intended for students who are already experienced at programming on Unix or Linux.

For larger projects a Unix utility called make may be used. make uses a file in the current directory, by default called makefile or Makefile, which must be written by the programmer, that contains instructions on how to compile a project that may contain several files.

A fairly simple Makefile might be:

```
all:  myproject

myproject:  myfile1.o myfile2.o myfile3.o myfile4.o
            gcc -o myproject myfile1.o myfile2.o myfile3.o myfile4.o -lm

clean:  rm *.o myproject
```

Makefiles are not free-format. The lines starting gcc and rm must each start with a Tab character, not spaces. After creating the Makefile, to build the project simply type make on the command line. The make program will look for a valid target in the Makefile. A target is a user defined keyword at the beginning of a line and which ends with a colon.

In our simple Makefile the targets are all, myproject and clean. To be useful, at least some of the Makefile’s targets must contain rules. Rules are simply a list of Unix commands (often compilation commands) that should be executed if that target is invoked. As stated above, the command lines in a rule must be prefixed by a Tab character.

By default simply running the make command will invoke the first target, in this case all. If we want to pick a specific target to run then we can type make targetname eg. make myproject.

On the same line as the target is a list of dependencies. These are files or other targets that this target ‘depends’ on.

In our example the target all only depends on the target myproject\(^1\) which in turn depends on the files myfile1.o myfile2.o myfile3.o and myfile4.o. Note that these end in .o and not .c. This is because they refer to the object files which a compiler generates when compiling a program with multiple source files.

Since all only depends on myproject we would achieve the same result by typing: make myproject as we would by typing make.

During compilation every source (.c) file gets compiled into an object (.o) file and then any necessary libraries, e.g. the math (-lm) library, are linked in to produce the final executable. During the make process make will compare the time and date of the object files to the source files and recompile any object files that are out of date and then build the target using the target’s rules (the command lines following the target and its dependencies). You will note also that the files here are also object files. This is done for speed since the .c files are already compiled into .o files we only need to combine the object files with the libraries. If we had put .c files here then each .c file would be recompiled!

The more observant of you will note that there is no rule to compile .c files into .o files. This is because make has built-in rules that do this for you so you don’t have to worry about it. Note: The command gcc -c myfile.c will create myfile.o.

In our example, the clean target is special because it does not do any compilation. By typing:

```
make clean
```

you can clean up the project i.e. remove the object and executable files. This is a nice simple way of tidying up your working area.

---

\(^1\)In this case the all target isn’t particularly useful, because it does nothing more than its dependency myproject. However, in more complicated Makefiles the default target will typically depend on several other targets and/or files.
Debugging Code

C

C.1 Understand Your Code

A program that produces nonsensical results contains a logic error. This is distinct from a syntax error, which is invalid C that causes the compiler to reject the program. It is often the case that logic errors can be fixed (i.e. the code can be debugged) merely by inspecting the code; no external debugging tool is required. Run through the code line-by-line, keeping track of what happens. Are the variables being used before you’ve given them an initial value? Does a variable get assigned the value you might expect? When your code meets a branch such as an if or select statement, does it follow the path that you would expect? Try drawing a flow chart of a program that solves the problem you’re working on, then verify that your code can be represented by that flow chart.

Simply by verifying that the code does what you set out to do in this manner can solve a large number of problems. Other situations such as garbage data or unexpected quits (often accompanied by messages such as “Segmentation Fault”) may not be so easy to fix. For this kind of bug it is appropriate to consider using debugging software.

C.2 Debuggers

There are a variety of C language debuggers available for the world’s various operating systems. The one we shall consider here is the GNU debugger, gdb as it is freely available for a wide variety of systems including Unix, Linux, Mac OS X and Windows. Where graphical debuggers such as ddd or XCode are available, these are simply front ends for gdb so the techniques described here will still be valid. Other debuggers may differ in their usage, but the concepts remain consistent. The instructions on using gdb will assume that you are using it from the command-line or Terminal on a Unix or Unix-like system.

C.2.1 Getting GDB

The Unix network in the computing lab. already has gdb installed and it may be invoked from a Terminal window. All modern Linux distributions have gdb but it may not be pre-installed; please check the documentation for your distribution. Mac OS X users need to install the XCode Tools (Developer Tools for OS X 10.2 or earlier) which include gdb. If you are using Windows, gdb may be obtained as part of an Integrated Development Environment.1 The latest version of gdb and more information about it may always be found at its website, http://www.gnu.org/software/gdb/gdb.html.

C.2.2 Using GDB

Compiling your code for GDB

The code that gcc produces by default is unsuitable for effective debugging, because it doesn’t contain debugging symbols. These symbols are hints to the debugger to allow it to determine what is going on in the program, and to relate this to the source code you supplied to the compiler. They consume significant space (the “Hello, world!” program increases in size by 26% on one system when debugging symbols are added), hence not being enabled by default. To compile in debugging symbols, pass the -ggdb symbol to gcc on the command line:

% gcc -ggdb -o filename filename.c -lm

1The IDE we recommend is Dev-C++; see
   for more information
Notice that the `-O` option, which can be used for code optimisation, is not provided to this invocation of `gcc`. Optimisation works by reordering code to be run more efficiently; it is not guaranteed that the debugging symbols generated from your C program will map correctly onto an optimised executable.

Running your program through GDB

To be able to fully inspect and modify your program while it is running, it must be launched from within a `gdb` session. From a Terminal window, invoke `gdb` with your program’s name (without the `.c` extension) as the argument:

```bash
% gdb filename
```

This will launch `gdb` and cause it to load your program and its symbols. You can now begin debugging. At most points during execution of `gdb`, you can type “help” to get a list of help topics. Once you have set up the debugger as you see fit, typing “run” will commence your program.

Breakpoints

It is often useful to stop your program executing at a particular point. Possible reasons include inspecting variables, stepping through portions of code a line at a time or even modifying variables while the program is running. This can be achieved by setting a `breakpoint`; the debugger will halt execution when it reaches a breakpoint and prompt for further action. Breakpoints may be set either at the beginning of a function by referring to its name:

```plaintext
(gdb) break function_name
Breakpoint 1 at 0x105a8: file filename.c, line 28.
```

or by specifying the line number:

```plaintext
(gdb) break 7
Breakpoint 2 at 0x10680: file filename.c, line 7.
```

The source code can be listed along with the corresponding line numbers using the “list” command, which shows the ten closest lines to the current execution point. Observe that `gdb` gives each breakpoint a number; the currently–set breakpoints may be observed by using “info break” and a breakpoint with number n may be removed with “delete n”. A one–stop breakpoint (e.g. in a `do...while` loop, when you only want to see what’s going on once around) can be set using the “`tbreak`” command which works the same way as break.

Inspecting your code using GDB

Your program is stopped, either because you have not yet run it, it has quit (whether in an expected way or not) or because it has reached a breakpoint. The following are a list of useful `gdb` expressions that can be carried out; see the online help for more information about them.

```plaintext
cont — continue execution.

step — continue, stopping at the next line of source code. The `step` command can be given an optional numerical argument (i.e. “`step n`”), to make it stop after n lines of code.

next — continue, stopping at the next line in the current frame. This is similar to `step`, except that if the line to be executed contains a function it will trace right through it, rather than stopping at the first source line within that function.

backtrace — prints the current “call stack”: that is a list of functions that the computer has had to enter in order to get to its current position. This allows you to not only see where the program is, but how it got there; for instance if your code died within a function `my_buggy_function()`, the call stack will allow you to find out from where `my_buggy_function()` was called.
```
set — this is actually a large and powerful collection of debugger functions under the same name, see the online help (by typing "help set") for much more information. Briefly, the set command can be used to modify all sorts of parameters while your code is running, including variables as shown in the sample gdb session below.

leeg@leegion:~>gdb simple_printfs
GNU gdb 5.3-20030128 (Apple version gdb-330.1) (Fri Jul 16 21:42:28 GMT 2004)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
(gdb) list
1     #include <stdlib.h>
2     #include <stdio.h>
3     
4     int main()
5     {
6     int a=3;
7     printf ("a = %d \n",a);
8     printf ("a = %d \n",a);
9     exit(0);
10    }
(gdb) break 8
Breakpoint 1 at 0x1dc0: file simple_printfs.c, line 8.
(gdb) run
Starting program: /Users/staff/leeg/simple_printfs
Reading symbols for shared libraries .. done
a = 3
Breakpoint 1, main () at simple_printfs.c:8
8    printf ("a = %d \n",a);
(gdb) set variable a=12
(gdb) cont
Continuing.
a = 12
Program exited normally.

C.2.3 Using GDB within Xcode

Xcode is an “Integrated Development Environment” and provides access to gdb functions without having to leave the graphical environment. Launching your program in the debugger is achieved by holding down the “Build and Run” button, and selecting “Build and Debug” from the pop-up menu. Setting breakpoints is performed by clicking on the margin in the code editor next to the line of code you want to break at. Xcode allows advanced debugging practices, such as logging when a breakpoint is reached but continuing execution and so on. For more information, see the document “Running in the Xcode debugger” in the Xcode online help.

C.3 Profiling and code optimisation

C.3.1 Should the code be optimised?

The programs that you will write for computing course practicals are all certainly too small for optimisation (rewriting the code to make the program run faster) to be a worthwhile effort; taking a thirty-second program and spending two weeks optimising it until it runs in ten seconds still means that it took two weeks longer. When writing larger projects (especially large numerical simulations as are prevalent in many fields of Physics) it certainly can be worth trying to improve its speed. Bear in mind the quote of Tony Hoare - premature optimisation is the root of
all evil. Without knowing how your program behaves you could easily spend a long time squeezing a little extra performance out of one function when the real benefits are to be made elsewhere.

Optimising techniques should not be considered if the code is not complete, and fully tested and debugged. But most importantly optimisation should not be undertaken if the code already runs fast enough. A simple point but one too easy to overlook - the optimised code could quite possibly be much harder to read or debug than the original. Bear in mind that the C compiler has a -O flag that will do certain optimising techniques without having to change your C source code.

C.3.2 The gprof code profiler

An example of a code profiler is gprof, available for most platforms and installed on the Computing Lab. systems. Note that most of the projects you will have been creating for the exercises in this book are much too simple for profiling to be effective. The way gprof works is to identify how many times a program enters a particular function and how long it spends in that function, and the exercises here typically require only one function (main) to be supplied by you. More advanced projects can benefit from code profiling.

To use the profiler, you first need to compile your code with profiling support:

```
gcc -pg programname.c -o programname
```

Now run your program from the command–line. Because it has been built with profile generation code, it will create an additional output file gmon.out containing the statistics. This can be read with the gprof command:

```
gprof programname gmon.out
```

The output of this command is a list of functions and the time spent within each function, first in order of decreasing time and then ordered by parent–child relationships (i.e. which function called which). The online manual page for gprof (see Section D.1) explains more.

C.3.3 Profiling in the Xcode tools

Xcode itself does not provide the ability to profile a piece of code, but several tools are distributed with Xcode which can achieve this. The document “Developing a Software Product with Xcode” in the Xcode online help describes all which are available; most are concerned with Object-Oriented programming, code for interacting with the graphical environment and other advanced topics which are beyond the scope of the C course. However, the Saturn utility (located in /Developer/Applications/Performance Tools/CHUD/) provides a graphical output of the same information that gprof provides, although Saturn’s output is easier to comprehend. To use Saturn the program must be compiled with the gcc invocation described in section C.3.2.

More advanced profiling utilities are provided in /Developer/Applications/Performance Tools/) including Sampler, which periodically stops your program and records its state; and Shark which can do the same thing for the whole system. Both of these tools have extensive online documentation, as well as web–based information at http://developer.apple.com/.

72 Appendix C. Debugging Code
Further Reading

D.1 Unix Manual Pages

Unix systems such as the Macs have an online manual system referred to as the manual pages, or frequently “manpages”. The manpages have a wide scope, covering user commands, formats for commonly-found Unix configuration files, C programming functions\(^1\) and much more. Manpages are primarily used as a source of reference information; e.g. looking up the invocation for a library function such as printf(). They are stored on the system as ASCII files marked up using roff, which is a formatting language similar in scope to \(\LaTeX\); you can view them using the \texttt{man} command in a terminal window.

D.1.1 Using the Manual Pages

References to manual pages are often given by their name followed by the section in the manual in which the page appears; for instance, the manual page for the C compiler would be referred to as gcc(1). Table D.1.1 provides the list of sections found in the Darwin\(^2\) online manual; the list is likely to be very similar on other Unix systems such as Linux. Note that some manpages have the same title – for example, there is a user command printf(1) as well as a C library function printf(3).

D.1.2 Manpages and Xcode

You will have noticed that when you create an Xcode project called, e.g. “hello”, one of the files that gets created is “hello.1”. This is the manual page for your project; if you were inclined you could edit this file to document your program. How to do this shall not be covered here; there are many websites which discuss roff documentation and the template file Apple provides should give some hints about how to proceed.

You can view the manpages for standard C library functions from within Xcode; from the “Help” menu select “Documentation” and type the name of a function into the search field in order to find its manpage. Note that the Xcode documentation contains material for much more than just the standard C API\(^3\) but the manual pages all have the phrase “man page” in the Parent field of the search results, so you should be able to see which files you are looking for.

\(^1\) Functions or interfaces in other languages – such as Perl and Tcl – may also be found depending on the system.

\(^2\) Darwin is the Unix operating system upon which Apple’s Mac OS X is based.

\(^3\) Application Programming Interface

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tools and Utilities</td>
</tr>
<tr>
<td>2</td>
<td>System Calls and Error Numbers</td>
</tr>
<tr>
<td>3</td>
<td>C Library Functions</td>
</tr>
<tr>
<td>4</td>
<td>Kernel Interfaces</td>
</tr>
<tr>
<td>5</td>
<td>Unix File Formats</td>
</tr>
<tr>
<td>6</td>
<td>Games</td>
</tr>
<tr>
<td>7</td>
<td>Miscellaneous Information</td>
</tr>
<tr>
<td>8</td>
<td>System Manager’s Manual</td>
</tr>
<tr>
<td>9</td>
<td>Kernel Programmer’s Manual</td>
</tr>
</tbody>
</table>

Table D.1: Table of contents of the Darwin reference manual
D.1.3 Manpages and the Terminal

From a terminal window (or an SSH connection), the command:

$ man printf

would display printf(1), to view printf(3) you would need to type:

$ man 3 printf

So far, it looks like man(1) is fine as long as you know what you’re looking for, but often you don’t even know that. For instance, you may know that you want a function for calculating sine, but can’t remember what it’s called. In this case you want to do a keyword search:

$ man -k sine

acos(3) - arc cosine function
acosh(3) - inverse hyperbolic cosine function
asin(3) - arc sine function
asinh(3) - inverse hyperbolic sine function
cos(3) - cosine function
cosh(3) - hyperbolic cosine function
sin(3) - sine function
sinh(3) - hyperbolic sine function

Finally, it is possible to get man(1) to produce PostScript output suitable for sending to a printer.

$ man -t fopen > fopen.ps

D.2 Bibliography

The following books or online resources contain useful information about the C language, numerical programming and programming for Unix. There are many such books available and they vary not only in quality but whether they are to the individual’s taste – the best approach is to go to a library or bookshop and read various books on a topic, and use the one(s) you get on with the best.

D.2.1 C reference guides

The C Programming Language 2nd Edition, Kernighan & Ritchie

The classic reference work on C, written by the authors of the language. The latest edition covers the most common variant of C, known as ISO/ANSI C or C89. The C language has been revised since then and the latest version of the standard is ISO C99; in practice there are few differences between the two revisions of the standard although C99 features some extensions over its predecessor. This course teaches C that complies with C99, but in practice the differences are small and many C89 compilers support C99 features as extensions. K&R, as it is commonly known, is widely regarded as a good book from which programmers can learn C, and not so good as an introduction.

The Standard C Library, Plauger

The companion to Kernighan & Ritchie, this book covers the standard library functions in a style similar to that used in the above reference. It includes excerpts from the standard and full code for standard library functions.
C Pocket Reference, Prinz & Kirch-Prinz
O'Reilly 2002 ISBN: 0596004362

A complete and comprehensive – but terse – reference to the C language and the standard library, including the modern ISO C99 variant. Also covers the C preprocessor.

C in an Nutshell, Prinz & Crawford
O'Reilly 2005 ISBN: 0596006977

The ‘bigger brother’ to Prinz’s pocket reference, this book also covers the C99–specific portions of the C language, as well as the standard library and the popular GNU C compiler (which is used by Xcode). It is both complete and discursive, containing many examples including examples of common mistakes.

D.2.2 C language tutorials

ANSI C: Problem Solving and Programming, Barclay

This book covers the entire C language and many library functions and is liberally peppered with example code, case studies and exercises for the reader. It covers additional subjects such as the preprocessor, Unix programming and revision control systems to some depth.

O'Reilly 1997 ISBN: 1565923065

Unlike the authors of many other C tutorials, Steve Oualline not only impresses the importance of writing to the language standard (C89, in the case of this book) but also of style, and of pragmatism above rote–learning intricate language rules. A chapter on the design process is included. The discussion is peppered with examples, exercises and simple debugging questions.

D.2.3 Numerical Programming


Numerical Recipes is considered by many to be the reference tome on numerical methods, as it has a comprehensive and detailed coverage of the mathematics behind a wide range of problems in scientific computing. That said, the authors are FORTRAN programmers by trade and the C code in the book is thought by some to be substandard as a result; the code also relies on the authors’ own data types defined to look like FORTRAN data types. Numerical Recipes is good for advanced readers who want to read about the maths involved in numerical recipes, and then write their own code to implement the maths.