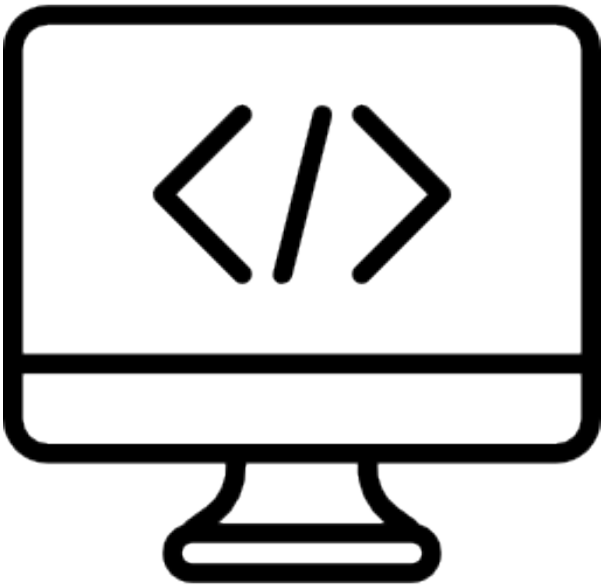
C PROGRAMMING

Practical course handbook



V.H. Belvadi

POSTGRADUATE STUDIES IN PHYSICS

YUVARAJA'S COLLEGE

CONTENTS

1. INTRODUCTION

Programming is an important skill for any physicist. Several computational techniques are employed in both theoretical and experimental work in physics that require the use of computers, access to terminals, and communicating with machines using carefully worded **commands**. While every single programming language in existence cannot realistically be taught in any physics course, it *is* possible to introduce students to the general workings of most computer-oriented languages. This is primarily because most languages take a similar approach towards writing a program even if they end up differing in terms of the precise forms of their instructions, called their **syntax**, and the efficiency with which they let the programmer achieve various end results.

Throughout this handbook anything that must be typed, including commands and code, has been set in a `highlighted monospaced font`. Other references to the system are set similarly but `without highlights` and all `syntaxes` are boxed. Certain words have been emboldened: these are programmers' jargon worth paying attention to and, in nearly all cases, their meanings will be apparent from the context in which they are used. The best, and perhaps the *only*, way to learn programming is to actually **write** programmes and **execute** them on your computer. And fail. Then keep going over the program, **debugging** it and retrying it until it runs successfully.

The aim of this course is not so much to teach C programming as it is to familiarise students with programming itself. Once you understand how you can communicate with a computer using a set of instructions, i.e. a **program**, learning other programming languages simply becomes a matter of learning new ways of writing similar instructions. Some languages are designed to serve specific purposes while others are general-purpose. Being one of the oldest languages, and being a precursor to most of today's modern ones, C is a natural starting point. By the end of this course you will *not* have learnt all that C has to offer, but you will have enough of a foundation in basic programming to be able to teach yourself different or even more advanced languages.

1.1. HISTORY

Brian Kernighan and Dennis Ritchie created the C programming language (and released the first edition of their book by the same name) during the 1970s. It was based on existing but somewhat primitive languages such as B and Basic Combined Programming Language (BCPL) but was powerful and robust enough that it has lasted to this day. The language and its syntax were standardised in 1989 by the American National Standards Institute (ANSI).

Today C is the language in which most software is written across various fields, including entire operating systems like unix and macOS, and nearly all software from Adobe Systems. Physicists use C for various purposes including modelling astrophysical systems, studying hydro-

dynamics, lattices and other molecular and atomic systems, general simulations, experiments that require data collection at lightning speed (such as the Large Hadron Collider) and, at the end of it all, to solve large equations with large arrays of variables. When we say someone works in C, we often also refer to one of the derivatives of C, such as C++, Python etc., all of which share a commonality with C in the fundamental manner in which a program is converted into instructions although, as we stated earlier, the specific approach and the exact format of instructions themselves may change from one language to another. Think of C as a car; these languages would then be more comfortable, faster cars — but cars nonetheless.

1.2. USING THIS HANDBOOK

One can keep talking for hours about C and its development and how it is worked on, but all of this is better learnt as we go. Almost all operating systems offer support for C out of the box but it always helps to have additional programmes that make life easier: some offer **syntax highlighting**, letting us read code better at a glance but styling different types of code differently; others offer auto-completion; still others offer detailed error analysis to make **debugging** easier. At the core of all this, however, the language itself remains the same and this is what we will focus on. That is, instructions in this handbook will remain as free as possible from workflows specific to third-party compilers. Students opting for such alternative solutions may have to help themselves but that should not be difficult. As far as the code and the knowledge required to write your own programs as part of this course goes, this handbooks covers all of it.

Use this handbook in order since each new section builds on the last. Each of sections 2–8 explores a particular idea, explains related syntax and gives example problems. Type in and execute these example problems on your computer, alter various statements in them and see what effect that has. Experimenting is often the best way to learn.

Towards the end of some sections are additional exercises with problems designed to help you understand the concepts explained in the preceding sections. Care has been taken to ensure that most programs are related to ideas in physics and mathematics which you have likely already come across during your previous years of study. Mixed among these exercises is a selection of example problems, one for each class of problems you may be asked for your semester exams or internal assessments, that has been asterisked. Solutions to these problems have been provided in section 10 but try not to sneak a peek. Again: execute as many problems as you can because programming, like physics in general, cannot be learnt merely through reading or rote memorisation. Think of all this as an extension of the problem solving you normally do in physics.

Working through section 7.3 is optional if you are running short on time in your computing lab sessions. Section 9 offers suggestions on reading material to continue your study of pro-

gramming beyond this course. There is also one final question there that covers most of the work you would have done by the time you reach the end of this handbook. Motivated students may, optionally, want to attempt this, and, if so, will have to treat section 7.3 as compulsory reading.

2. OVERVIEW OF A C PROGRAM

Before we discuss the softwares you will be using to write and execute your C programs, let us take a moment to understand what exactly we will be doing over this process.

A program is a set of instructions you give to a computer to follow. Each instruction is a command and various commands are cleverly grouped together to draw the required results or **output** from the computer. The main purpose of this, of course, is to exploit the fact that computers can be faster than humans at following instructions (and arguably more competent) as well as at computing and otherwise memorising and handling large data.

The simplest instruction one can give a computer is to ask it to display something on the screen. This is called **printing** and is written as

```
1   printf("Hello, world!");
```

There are a few things to note here that will help any further programming you do:

— Notice the semicolon `;` at the end of the line. In C programming, the semicolon is a **terminator**. It tells the computer that the command has ended.

— The command itself is `printf` and it tells the computer to display or **print** something on the screen. This 'something' is called an **argument** and the `printf` command accepts an argument which it will simply display on the screen.

— The `printf` command has a **syntax** that looks like `printf(" ");` and anything inserted in-between the inverted commas `" "` will be **printed** on your screen.

— Entering any part of the command wrongly will result in a broken program with **buggy** code. You will then have to debug and **execute** the program again. You may find yourself doing this repeatedly: do not lose heart because this does not mean you are a bad programmer, it is often just how things work.

Entering the above line of code into your computer, however, will not do anything. This is simply one command and we still have to make a program out of it. We will now add a few other lines that will tell the computer what to execute and will, in fact, work like a program. The output of this program, as expected, is simply to display the words 'Hello, world!' on your screen.

A program is always started by calling **header files**. The `stdio` header file is mandatory for all programs and defines for the computer a lot of commands and keywords we will be

using in our program. As we learn more about C we will come across several other header files too which will help us build more complex programs. Such files are **called** using the `#include < >` command and the name of the header file is given as the argument with the extension `.h` which identifies the file as a header file.

Consider the following lines of code next:

```
2   #include<stdio.h>
3
4   int main() {
5
6           printf("Hello, World! \n");
7
8           return 0;
9   }
```

All contents of a C program are included in functions. We will discuss these in detail later. Notice at this point, though, that parentheses and braces segment a program into a number of **tokens**. We usually represent each token by a line, but you need not worry too much about this. All that is important to understand at this point is that whitespace has no meaning to the computer itself in C and is only meant for humans to read. That is to say, the above program can just as well be written as follows:

```
10  #include<stdio.h>int main(){printf("Hello, World!\n");return 0;}
```

The output of the program would be the same as before but it would be hard for us to understand a program, especially if it is long and complex. Note, though, that some whitespace *does* matter: writing `intmain()` is *not* the same as `int main()`. In any case, this is what a C program *looks* like and most C programs you will come across in this handbook (and in your later studies) will look like necessarily more complex forms of this.

3. COMPILING AND EXECUTING

There are several ways to execute or run a C program. Once the contents of a program are typed, they must be saved into a file with the extension `.c` on your computer. Say we save our example program from earlier into a file called `hello_world.c` located at `/folder/path/hello_world.c` on your computer. Try not to have spaces in your filename; use underscores or hyphens instead.

Most Integrated Development Environments (IDEs) have a simple *run* button that can execute your program in one click. Since the exact approach to this varies from software to software, we will not be going over it. Besides this any C program can always be run via the **Terminal** available in almost any computer and it is this method we will overview now.

In the lab you will be working on a unix-based operating system and opening the Terminal will be like opening any other app. Once you are in the Terminal you should see something that looks like this:

```
11  computer@name:~ username$
```

The `$` symbol is called a **prompt**. You will find various prompts based on the terminal environment you are in. Ideally, it is the dollar prompt you should be seeing as soon as you open a Terminal window. From this point on, for simplicity and clarity, we will be omitting everything *before* the dollar prompt and only mentioning the commands you will have to enter *after* the prompt. Make sure *not* to type in the dollar prompt `$` itself at any point.

Now navigate to the folder in which you have saved your C program. This is done using the `cd` command, which stands for **change directory**. The syntax is `cd /full/folder/path` to point to the directory you want to enter. If you are ever in doubt, you can simply type `ls` after the dollar prompt to view a list of files and folders and then type `cd folder_name` to enter a particular folder.

```
12  $ cd /folder/path
```

Now that you are in the required folder you should be in a position to execute your C program. However, executing a `.c` file is the *last* step. First we need to convert the `.c` file from its human-readable format to a machine readable file, called an **executable**. This is called **compiling**[1].

[1] In the lab all systems have the gnu C compiler preinstalled. Instructions to install this on your own system can be found at `https://gcc.gnu.org/install/` from the Free Software Foundation, Inc.

POSSIBLE ERRORS

If you are unable to locate your folder using `ls` or if you ended up choosing the wrong path, you can use the command `cd ../` to return to the immediate upper level. Note the space in that command since 'small' points like this make a huge difference in programming. In your terminal, as in line 11, the tilde `~` represents your home directory; typing in `cd ~` will take you back home and let you start afresh. Also remember that you can type in `clear` anytime to clean up the terminal window.

At this point make sure that you have saved the 'Hello, world!' program (lines 2 to 9) into a file called `hello_world.c` or do so if you have not. We can now compile a C program using the `gcc` command. This only applies if you are using gcc, which you will be doing in the lab. Other programs may use different commands, such as `clang`, instead of `gcc` but with the rest of the syntax untouched.

```
13  $ gcc -o exe_name file_name.c
```

where `exe_name` is the name of the executable file that will be created for your program. Usually, this is named the same as the `file_name` but you can choose anything you like. The command `-o` gives the terminal permission to open the file[2].

In our case we can compile our program using the command

```
14  $ gcc -o hello_world hello_world.c
```

If the compilation is successful you should not see any errors on your screen. If you do get errors, go over this section again and track down where you may have made a mistake. Spaces, semicolons and other symbols that are easy to miss are some of the most important ones. You can solve errors during compiling mathematics by appending `-lm` to line 14, telling the compiler explicitly to include the `math` library, e.g. `gcc -o hello_world hello_world.c -lm` especially on old systems.

Once compiled it is time to execute our program. For this we have a simpler syntax:

```
15  $ ./exe_name
```

The `exe_name` must match the name of the executable file that we created at **compile time**. Through these commands we are telling the computer that we have a `.c` file that it must compile

[2]Note the differences between the lowercase letter 'oh', written as `o`, the uppercase letter 'Oh', written as `O`, and the number 'zero', written as `0`. The uppercase letter has less rounded corners and the number is more of an oval.

to a machine-readable format and then execute to provide us an output.

Learning syntaxes is important in C as they provide a framework on which we can write down instructions/commands as we like. In this case our executable file is called `hello_world`, which is what we named it in line 14, so our execution command becomes

```
16  $ ./hello_world
```

This should then give you the output, which is simply the words 'Hello, world!' printed on your screen. By itself this program does nothing impressive, but you have now written your first C program successfully. The process of compiling and executing explained in this section applies to all your future programs too.

TIPS

Throughout this process, ensure that you are in the same folder/directory as the one in which your C program, i.e. your `hello_world.c` file, is saved. Also, you can combine the compilation and execution commands into a single command that will automatically be run both its constituents one after the other. Use `gcc -o exe_name file_name.c && ./exe_name` as your syntax.

4. VARIABLES, DATA TYPES AND RESERVED WORDS

4.1. PRINTING THINGS ON SCREEN

As in physics, a **variable** is a quantity that takes different values under different conditions. The manner in which a variable is treated in C is different, of course. As in any computer program, C allots a certain amount of space in your computer memory to every variable and uses an **identifier** to reference that memory location which holds the value or magnitude of that variable.

For example, you can call a variable `x` and specify its **data type** as an integer and C will allocate an appropriate amount of space in your computer memory to it and call that space `x`. Every time you refer to `x`, your computer now knows that you refer to the value stored in that particular memory location.

Consider this simple program to add two integers:

```
17  #include<stdio.h>
18
19  int main() {
20
```

```
21          int a=2, b=4, c;
22
23          c = a + b;
24          printf("The sum of %d and %d is %d \n",a,b,c);
25
26          return 0;
27  }
```

Let us see carefully what this program is doing, line-by-line (except line 26 which we will talk about later). Pay attention to new commands and syntaxes you will come across now:

LINE 17: This is the same header file we saw earlier in our `hello_world.c` program.

LINE 19: This is called a **function**. A function is a subset of commands in a C program. The syntax of a function is `data_type name() { }` and we have used the data type `int` (which we will discuss presently) and the function name, `main()` which is a standardisation: every C program starts with the first line of the `main()` function. Other functions can be called from within the `main()` function, but there must always exist a `main()` function which contains the heart of a program. We will see how this can be done in later programs. Note the lack of a semicolon at the end of a function.

LINE 21: This is known as the **declaration** and is where we first tell the computer how many memory slots we will need and what we will be using them for. Declaring variables is preferably done at the start of a program to prepare necessary memory locations. This is also where **identifiers**, or names, are assigned to our variables. We have also **initialised** variables `a` and `b` with some values.

The syntax for such a declaration specifies the data type and identifiers and any initial values as `data_type identifier=initialisation, identifier, ... , identifier;` and our declaration too uses this format.

Our `data type` is `int`, which stands for integer, and our identifiers are `a`, `b` and `c`, two of which we have **initialised**, i.e. given values to at the start of the program. Naturally then, `c` is *not* initialised since we will be adding `a` and `b` and assigning the value of their sum to be stored in the memory location identified by `c`.

LINE 23: This is a simple arithmetic operation. Any C program supports basic operations such as sums, differences, products and quotients out of the box. For more complicated

operations like powers, we must also include the `math` header file that defines other operations. Once this header file is called, b^a can be defined as `pow(b,a)` and $\sin x$ is `sin(x)` and so on.

In our program, this line tells the computer to add `a` and `b` and store their sum as `c`. So when we call `c` next time the computer simply gives us the sum `a+b` instead.

LINE 24: We have come across this line before in our `hello_world.c` program and we know that its job is to print whatever is stated between a pair of inverted commas. However, we have an new format here compared to last time: besides simple text we have a reference to `c` as well.

The syntax is `printf("Some text %d more text %f", identifier, identifier);` where the percentage symbol tells the computer that we are referring to a particular data type and the letter/s immediately after that are the **format specifiers** which tell the computer whether we are referring to an integer (`d`) or a decimal number (`f`) and so on.

The contents within the inverted commas are printed and the `%d` and `%f` are replaced by their corresponding identifiers mentioned in the same order and separated by commas *after* the main pair of inverted commas.

In our program we refer to three integers and therefore use `%d` to specify where they must be placed; next we specify `a`, `b` and `c` as our identifiers, in the order in which they should appear. That is, the first `%d` will be replaced by `a`, the second by `b` and the third by `c`. The output of this program will then be 'The sum of 2 and 4 is 6'.

Now try to run this program yourself. Feel free to also make changes and see what effects they will have. If you break the program, follow the errors and try to rectify them yourself.

4.2. ACCEPTING INPUTS FROM THE USER

Every value in a C program is classified into a **data type** which determines how much memory storage it is allocated. That is from the perspective of a computer. From a human perspective a data type is simply a determinant of the type of a variable: is it an integer, a fraction, a letter, a string of letters etc.

There are four basic data types in C which have further classifications beneath them. Here is an overview, but do not treat this as an exhaustive list:

1. Primitive data types
   (a) Integers
   (b) Characters
   (c) Short
   (d) Long
   (e) Float (decimals)
   (f) Double

2. Derived data types
   (a) Pointers
   (b) Arrays
   (c) Structured
   (d) Functions
3. Enumerated data types
4. Void data types
   (a) Certain functions
   (b) Certain pointers

Writing a one-sided C program is like being a physicist trying to work things out without maths: interesting, but of little consequence no matter how you look at it. One of the fundamental capabilities of a C program is to interact with the user; to take in some data and **process** it or **compute** something using it and to eventually produce some output.

Like 'print' was the term used to display data on the screen, **scan** is the term used to accept data from the user. The command used for this is `scanf` and, like the `printf` statement, the syntax of a `scanf` statement involves format specifiers and their respective identifiers, but this time to save data to memory rather than retrieve it. To accept various types of inputs use the syntax `scanf("%d %f %s", &identifier, &identifier, identifier);`. This saves inputs to the memory locations specified by their identifiers. The ampersand symbol (`&`) represents the memory location of an identifier rather than the identifier itself.

POSSIBLE ERRORS

Note that there is no location identifier for characters and strings. The **address-of** symbol `&` tells you the memory address of an identifier while typing just the identifier gives you its value. Make sure you are asking for the right thing since the computer simply does what you command it to do. The `scanf` statement essentially tells the computer to 'read a type of input and save it to the memory location named as a particular identifier'. This means if you expect the user to enter values separated by commas, the data types in your `scanf` statement should be `scanf("%d, %d",&a,&b);` also separated by commas. This statement accepts an input like `7, 8` and stores `7` in the memory location of `a` and `8` in the location of `b`, but it throws an error (or uses stray values) for an inputs the form `7 8` etc.

Why does the identifier for `%s` not have an address-of symbol before it? This is because C stores strings as an array of characters which means their addresses are already implied. You do not need to worry about exactly what this means just yet as we will discuss strings and arrays in detail later.

The following are some commonly used data types in C, and in most languages derived from it, along with their symbols, declarations and memory sizes in bytes:

`%d` Integer `int` 2–4 B `%ld` Long integer `long int` 4 B `%f` Float `float` 4 B

`%c` Character `char` `%lf` Double `double` 8 B `%s` String

4.3. COMMENTING AND DEFINING CONSTANTS

The enumerated data type `enum` is not listed here since it is a little different from the others. An enumerated data type lets you store custom-defined identifiers as a collection. Its exact behaviour is out of our scope of discussion, but it is worth knowing, for completeness, that an `enum` lets us declare a set of identifiers, all for an arithmetic data type, and handle them in batches.

For example, declaring `enum quarks { up, down, strange, charm, top, bottom };` lets us assign any one of these **elements** as the value of any other explicitly declared variable of the new data type `quarks`. Throughout a program then these **enumerators** are effectively treated as constants.

Here is a simple example:

```
28  #include<stdio.h>
29
30  int main() {
31
32      enum weekday { monday, tuesday, wednesday };
33      enum weekday today; // A 'weekday' variable called 'today'.
34      today = monday;
35
36     /*  An alternate way of declaring such
37      *  variables is to use the combined format:
38      */
39
40       enum weekend { saturday, sunday } that_day;
41
42     /* This means we now have a variable
43      *  called 'that_day' of type 'weekend'.
44      */
45
46      printf("Tomorrow is day %d of the week\n",today+2);
47
48      return 0;
49  }
```

The output of this program would be 'Tomorrow is day 2 of the week'. But, besides `enum`, there are some important observations to be made here, like in our last program, that will help us later:

LINE 33: All the text following the double slash `//` on this line will be ignored by the C compiler and is meant for the programmer's eyes only. Such texts are called **comments**. Think of comments as little notes you leave to yourself for future reference in a program or to other programmers to help them understand your program better.

LINES 36 TO 43: Everything between these lines constitute a **multiline comment**. Make a multiline comment by enclosing text between a slash followed by an asterisk `/* ... */` and the compiler will simply ignore these lines. As before, they are for programmers only. Note that the series of asterisks `*` typed at the start of each of these lines is *not* part of the syntax, but using symbols like this is common practise in programming and ensures that multiline comments are neatly set apart from the main lines of code.

LINE 46: The `printf` command here not only refers to the variable `today` to print it in place of %d but also specifies an arithmetic operation right inside the `printf` statement, i.e. it asks the computer to print not the value of `today` but the value of `today` with `2` added to it.

This is an efficient shorthand and saves us a line: we do not strictly have to add `today+2` separately. In the same vein, observe that line 23 could have simply been included in line 24 and we need not have declared a third variable `c` at all. This type of improved efficiency is a huge part of what makes programs run faster.

TIPS

Note how our main function too has a data type `int` associated with it. What does this mean? In short, this refers to the fact that our `main()` function **returns** a value of type `int`, but we know this has not been true so far. Our `main()` function has *not* returned any data at all besides displaying an output as commanded. It is for this reason that we specify (as in lines 26 and 48) the code `return 0;` suggesting that the `main()` function is, in fact, returning nothing. Although it might seem tempting to simply call our function `main()` without specifying a data type, this is invalid syntax.

Besides declaring constants in `enum` we can explicitly define individual constants too. Of course initialising a variable also equates it to a constant, but then we can redefine the same variable and overwrite its value, so it is *not* truly constant. However, using `#define name value` we can define a proper constant for use in our program.

The definition of a constant can be (and usually is) done soon after we call our header files, *outside* our `main()` function. We can also define other variables this way to make them available **universally** for use across all functions; else, if declared inside our `main()` function, the variables remain valid only within the `main()` function. Do not worry too much about this as we will discuss this in detail later.

Consider a simple program in which we use ideas we have discussed so far. The program accepts the value of a radius from the user and computes the area of a circle and volume of a sphere of that radius.

```
50  #include<stdio.h>
51  #include<math.h>
52
53  #define pi 3.1415926536
54
55  // Compute the circular area and spherical volume, given a radius
56
57  int main() {
58
59      float radius, area, volume;
60
61      printf("\nEnter the radius:\t");
62      scanf("%f",&radius);
63
64      area = pi * pow(radius,2);
65      volume = (4.0/3.0) * M_PI * pow(radius,3);
66
67      printf("\nArea = %f",area);
68      printf("\nVolume = %.2f\n",volume);
69
70      return 0;
71  }
```

Understanding and executing this program should give you enough of an idea to work through the exercise that follows. But first, here is a quick look at some important lines:

LINE 53: Here we define the constant `pi` for use later in the program. Note that this is for the sake of our example only; the `math.h` header file predefines a constant called `M_PI`

which carries a more accurate value of pi. We have used this in line 65.

LINE 59: Foreseeing that the user may enter a fractional value of radius and not necessarily an integer we use the `float` data type here for our radius, area and volume.

LINES 64 AND 65: Note that `math.h` allows us to use `pow(radius,2)` and `pow(radius,3)` instead of the long-winded `radius*radius` and `radius*radius*radius`.

Also note our use of the self-defined constant `pi` in one line and the predefined constant `M_PI` from `math.h` in the next line.

Lastly, since we are dealing with `float` type variables, our fraction cannot be just `4/3` but must also be of float, i.e. fraction/decimal, type explicitly, meaning we must specify that there is nothing beyond the decimal point (at least in the numerator or denominator if not in both) as `4.0/3` or `4/3.0` or `4.0/3.0` so as not to end up with incorrect computations.

LINE 68: While displaying the result of our area we simply used `%f` for the identifier `area` but here, for the `volume`, we use `%0.2f`. The syntax `%wf` carries the **field width specifiers** and tells the computer to restrict the value of `%f` to `w` width.

More specifically, `w` is made up of two components `w.p` meaning the field width and precision. The precision describes the number of digits after the decimal point and the field width is the number of digits, counting from the right, that the number should occupy.

For instance, if `radius = 3` our volume is about `113.09734` and if we use `%0.2f` to display this, the volume will be rounded off to `113.10` and if we also specify a width of seven by stating `%8.2f` then the same number is displayed as `  113.10`, i.e. the number is displayed in a 'field' with eight blanks to be filled, counting from the right: zero occupies the first, one occupies the second, the decimal point occupies the third etc. so that the five digits and the decimal point together occupy six spaces and leave two unoccupied ghost spaces in the thousands' and ten-thousands' places.

TIPS

You may have noticed that we have been using the code `\n` a lot within our `printf` statement. In the program below we use a new one `\t` as well. These are called **escape sequences**. The command `\n` is called a **new line character** and tells the computer to end a line at that point and start a new one. The `\t` command is a **horizontal tab character** and tells the computer to give a tab space at that point. We will be using them a lot henceforth to better arrange the display of data on our screen and they may confuse you at first. The best way to work around this is to execute programs and see the effect they really have. Try deleting one of the `\n` or `\t` commands and executing your program and see what effect it has. Neither will break your code; they will merely change how it is displayed on screen.

So far you have learnt how to display something on screen, accept inputs from the user, perform basic arithmetic operations, define constants and handle certain data types. A set of programs are given below as an exercise for you to work through. We will overview data types again before discussing characters and Boolean data types.

EXERCISE 1

1. Write a program to accept all required data in mm from the user and calculate the reading of a screw gauge in cm correct to three decimal places.

2. Write a program to accept an integer from the user and display the numbers before and after it. You are allowed to declare only one variable.

3. The syntax `sizeof( )` takes a data type (`int`, `double` etc.) as its argument and is an identifier that will retrieve for you the allowed memory size of that data type. Print its value as a data type `%lu` and verify if the byte lengths given above for `int`, `char` etc. match those on your system. You may have to use the `limits` header file.

4. Accept the lengths of the two shorter sides of a right-angled triangle from the user and do the following: (a) use Pythagoras's theorem to find the hypotenuse, and (b) use the sine rule to find the two unknown angles.

5. Ask the user for the coördinates (coefficients only, to be entered as `(x, y, z)` into the program) of two vectors and (a) add them up, (b) subtract them, and (c) find their dot product.

TIPS

First write down the various steps involved in your strategy when trying to solve a problem; write down what variables you will need, of what data types they are, and exactly what you will do with them going from accepting input and assigning values all the way to computing, analysing and presenting the required output. Write all this verbally, stepwise and use it as an outline to write your program: this method of strategising of your program is called an **algorithm**. For help in writing one, see 5.1.

4.4. CHARACTERS AND THE BOOLEAN DATA TYPE

We saw that an integer data type is a basic data type that stores integral values, a float data type stores fractions/decimals and long integers and doubles allot more memory to store longer versions of integral and fractional values respectively. We also saw that, ultimately, all data types are converted into bytes that a computer can understand when an executable file is created. For this reason, a `char` data type is sometimes also classified as an 'integer' data type although calling letters integers does not make immediate sense.

However, look at it from the perspective of a computer: storing a single character takes a single byte. This is the only constant across generations of machines. An integer, for example, can have various storage spaces depending on the architecture of a system (for example, 32-bit and 64-bit CPUs can potentially allocate different storage amounts to the same data type). But, more importantly, every character is stored as an ascii-equivalent number[3]. The decimal ascii value of `S` is 83, so the character `S` is stored the same as the one-byte integer, 83, in memory. Whether this is a number or it must be converted into its equivalent character will be decided by the compiler when the program is being run depending, of course, on the program.

Absolutely every character you can type or print on your screen, be it a letter, a number, or a symbol, has an ascii value associated with it. This allows us, comically sometimes and usefully at other times, to 'add' to letters. Try this out in a program: the ascii values of `a` and `c` are 97 and 99 respectively. What happens if you declare two characters as `'a'` and `'c'` try to print their sum?

Declaring a character is the same as declaring any other data type, but initialising them must be done inside single inverted commas: `char identifier, identifier='initialisation';` For example, we can declare a character called `char a, b;` where `a` and `b` are identifiers and *not* the values stored in the characters themselves. Following our little experimental program from the last paragraph we can initialise them as `char a='a', b='c';` which means the iden-

[3]The American Standard Code for Information Interchange (ASCII) is a character-to-number conversion standard.

tifier `a` now stores the value of the character `'a'` inside it and the identifier `b` now stores the value of the character `'c'`. You can now try adding `a + b` to see what output you get.

Perhaps writing this differently will clear things up: the declaration give by the command `char first_letter, second_letter;` makes it clear that `first_letter` is *not* a letter but an identifier of the type `char`. As with any other data type, we can make it store a letter by initialising it: `char first_letter='a', second_letter='c';` and then add them to get an output: `first_letter + second_letter` which turns out to be an integer.

There is yet another data type (although not one you may be using a lot in this course) called the Boolean data type. This data type, as its name suggests, only allows one of two polar values: a plus and a minus, or a head and a tail, or, more specifically in programming, a one and a zero.

A Boolean data type, named after George Boole, has a special declaration in derivative programs like objective C as `bool identifier;` and the identifier normally takes the value `1` if it is true and `0` if it is false. C itself does not have a `bool` type and a simple `int` can be used equivalently and, in fact, is used more often.

That said the purpose of introducing the idea of a Boolean quantity here is to talk about the derived ideas of a Boolean condition and a Boolean statement that we will come across later. A Boolean condition is one that, like a Boolean variable, will be either true (`1`) or false (`0`). We use Boolean conditions to check whether relationships between variables satisfy certain requirements we are interested in or not.

4.5. KEYWORDS AND SYSTEM COMMANDS

There are a lot of words that C uses as part of its syntax that a user is *not* allowed to use as an identifier because doing so may lead to utter confusion as to whether something is a command or an identifier and, in turn, will almost certainly result in an incorrect output. Such prohibited words are called **keywords**. The following are keywords used in C that you cannot use yourself as anything but commands:

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

POSSIBLE ERRORS

C is a **case-sensitive** language, which means writing `M_PI` is *not* the same as writing `m_pi` or even using an identifier called `radius` is *not* the same as `Radius` and so on. Along the same lines, remember that initialising a character is done with single inverted commas (also called 'quotes' sometimes) and not double inverted commas: `'m'` and not `"m"`. We will later see that double quotes are reserved for strings.

You may recall those which you have already come across. You will come across more of them as we go. But remember that even keywords are allowed in two parts of your program: one, as an argument for the `printf` command, since that data will be displayed exactly as given and will not be processed except to check to format specifiers; and, two, in comments since the computer ignores comment lines.

The `system` **reserved word** (another name for keyword) can be used to issue commands to the terminal from within a C program using the syntax `system("command");` where the `command` is anything you want to instruct the terminal or system itself.

There is a particularly helpful **use case** for this. We saw earlier that we can clear the terminal window using the `clear` command after the `$` prompt. We can also say `system("clear");` *inside* our C program to have the same effect and clear our screen. It is advisable to give the first command to clear the terminal window *after* declaring and/or initialising variables.

Like the `\n` and `\t` statements this is more of a superficial feature in our program but starting with a clean window means users can get a better look and a better understanding of whatever is happening on screen. We will start using this command, albeit sparingly, in our code from this point onwards.

POSSIBLE ERRORS

Older systems may use `cls` or `clrscr` or equivalent commands. Most of these have been deprecated in modern standardisations of C. The same is true of the `conio.h` file itself. Do *not* use any of these. If you are unable to pass a `system("command");` directly, try calling the `stdlib.h` file.

EXERCISE Try writing a C program to list out all files and folders in the current directory.

5. DECISION MAKING WITH `if` AND `else`

5.1. ARITHMETIC OPERATORS

It is now time to introduce some basic intelligence into our programs. Think of yourself in a supermarket: **if** you have money in your wallet, you will buy a certain interesting-looking

article, **else** you will not buy it (or you will put it off for another day, but that is besides the point).

Such `if` and `else` statements exist everywhere in life and are just as common in mathematical and physical arguments. While solving a problem you can instruct the computer to execute certain set of commands `if` a condition is true, or **else** execute a different set of commands. The syntax is simple: `if ( ) { } else { }` where the condition is enclosed in parentheses and the commands in braces.

Let us now use this idea to write a small program that will check if a given number is even or odd. First, though, let us write an **algorithm** that will outline the strategy we will be following in this program. An algorithm is not a massively complex idea; it is simply a watered-down version of your program, written as steps, preferably with syntaxes, to make writing your actual program easier. Although algorithms were written by physicists as early as Euclid (see the exercise problem below) they were popularised in computer programming by the likes of the English mathematician, Alan Turing.

Now let us get back to our program to check if a number is even or odd. Our approach will revolve around the fact that even numbers are divisible by 2 and odd numbers are not. To check if a number is divisible by another, we use the **remainder-of** operator `%`. Writing `a % b` means the operator returns the *remainder* of the division of `a / b`. So if we check `x % 2` for some given `x` and if this turns out to be zero, then we can safely say that `x` must be an even number; else, `x` must be an odd number. Formally, this is what our algorithm would be:

PRINT Enter an integer

INPUT x

IF $x \,\%\, 2 = 0$ **THEN PRINT** x is an even number

ELSE PRINT x is an odd number

Note that although we use some keywords, the exact syntax is not always used in an algorithm. At the end of the day, write your algorithm however you find it comfortable to understand but try to follow a pattern similar to the one above.

The sole purpose of an algorithm is to make it easy for you to write your actual program. We can now use this algorithm to prepare a working program that we can execute on our systems. A few line-by-line observations follow.

```
72  #include<stdio.h>
73  #include<stdlib.h> // Called only for the |\verb+system("clear")
        ;+| command
```

```
74
75  int main() { // Check if a given number is even or odd
76
77      int x;
78
79      system("clear");
80
81      printf("\nEnter an integer:\t");
82      scanf("%d",&x);
83
84      if ( x % 2 == 0 ) {
85          printf("\n%d is an even number.\n\n",x);
86      } else {
87          printf("\n%d is an odd number.\n\n",x);
88      }
89
90      return 0;
91  }
```

LINE 79: The `system("clear");` command has been used here to make sure the output of the program looks clear and easy to follow as we discussed before. In general, however, `system( );` commands are best avoided as far as possible: they can be a security risk as they let you meddle with the system through a C program, and they make your program system-dependent as not all `system( );`. commands work on all systems.

LINE 84: The remainder operator, sometimes also called the **modulo operator**, is used here to check if 2 divides `x` fully in order to determine whether `x` is even or odd. This is all exactly as we discussed while writing the algorithm.

LINES 84 TO 88: There is one important thing to remember about using braces to group commands together, as we have done for our `if` and `else` statements here: if only one command exists, no braces are needed. Omitting braces may not seem like much, but in longer programs, every additional character that the computer has to read through will delay execution time and such single characters tend to add up to make a substantial difference.

In other words, the program would have worked just fine had we written lines 84 to 88 simply as lines 92 to 95 shown below:

```
92        if ( x % 2 == 0 )
93            printf("\n%d is an even number.\n\n",x);
94        else
95            printf("\n%d is an odd number.\n\n",x);
```

There are seven arithmetic operators: the sum `+`, the difference `-`, the product `*`, the quotient `/`, the remainder `%`, the increment `++` and the decrement `--` operators.

The increment operator increments the value of the existing variable by one, i.e. it adds one to the current value. The decrement operator does the opposite. To increment or decrement some variable use `variable++;` or `variable--;`. Try rewriting problem 2 from exercise 1 using the increment and decrement operators. Note also that the increment and decrement operators are simply shorthand for the lines `x = x + 1 ;` and `x = x - 1 ;` respectively.

5.2. JUMPING TO A LINE

You can move to a specific line in your C program using the `goto label;` command. You will first have to flag or label your line with some `label:` and you can then jump to it from any other part of your C program. Pay attention to the fact that there is a colon at the end of a `label:` and *not* a semicolon. A label must be placed at the start of a line. It will then refer to the command identified from that point up to the first semicolon the processor encounters.

5.3. RELATIONAL OPERATORS

Besides arithmetic operators, C also defines relational operators that help us check the relationships between two quantities. There are six relational operators, almost none of which need explanation: equal to `==`, not equal to `!=`, lesser than `<`, greater than `>`, less than or equal to `<=` and greater than or equal to `>=` and all of them have the same meaning as in physics and mathematics. Pay attention to spaces: `! =` is wrong, and `< =` is wrong too. Instead write these symbols *without* a space between them.

Relational operators see great use in defining `if` conditions. Before we see an example problem, note that the `=` symbol is *not* a relational operator and does *not* mean 'equal to'. Instead it serves as an **assignment operator** and writing `c = a + b ;` really means 'Add `a` and `b` and assign to `c` the value of their sum'. The command `c == a + b ;` is a **conditional statement** that checks the relationship between `c` and `a + b` and returns a value of `1` if it is true and `0` if it is false. In other words, it is Boolean.

Now it is time for another example program (see the following page). This time, however, no description of the problem will be given, just the program itself. By now you should be in a position to look at the code, *read* it and identify what it is doing. This program may seem fairly

complicated now, so here is its algorithm to help you out:

PRINT Enter an integer

INPUT x **[TRY]**

LET $y =$ integral part of x (i.e. only the digits before the decimal)

IF y != x **THEN GOTO [TRY]** (i.e. retry if the integral part of x is not equal to x, that is if $x \notin \mathbb{Z}$)

ELSE IF x % 2 != 0 **THEN PRINT** x is an odd number

ELSE PRINT x is an even number

First of all, notice how we have used the label `TRY:` to reference the input statement. Then we have used the `goto` command to return to that line.

Secondly, notice that our `if ... else` statement has an `if ... else` statement inside it. This is called a **nested if ... else statement** and allows us to check for complex parameters or, quite simply, conditions within conditions.

The syntax of the nested `if ... else` statement follows from the regular `if ... else` statement: `if ( ) { } else { if ( ) { } else { }}`. A couple of other ideas are explained in the usual line-by-line observations after the source code.

```
96   #include<stdio.h>
97   #include<stdlib.h>
98
99   int main() {
100
101      float x;
102      int y;
103
104      system("clear");
105
106      printf("\nEnter an integer:\t");
107  TRY:scanf("%f",&x);
108      y=(int)x;
109
110      if ( y != x ) {
```

```
111             printf("\nYou seem to have entered a fraction.\nPlease
                    enter an integer:\t");
112             goto TRY;
113         }
114         else if ( y % 2 != 0 )
115             printf("\n%d is an odd number.\n\n",y);
116         else
117             printf("\n%d is an even number.\n\n",y);
118
119         return 0;
120     }
```

LINE 101: We declare our variable, `x`, as a `float` type in preparation for some user entering a fraction into the program.

Try entering a non-integral quantity in our previous program to check if a number is even or odd and you will find that it breaks the program. This modification intelligently lets the computer give the user another chance to enter an integral if they entered a fraction in the first place.

LINE 107: We use the label `TRY:` to flag/label this line and return to it later. Although we skip back a few lines in this program, this technique can be used just as well to skip forward a few lines.

LINE 108: We originally declared `x` as a `float` type. If we want to check if the user entered an integer, we must make sure that the `float` variable `x` is equal to that part of `x` which exists only before the decimal point, i.e. the integral value.

To check if this is true, in this line we assign the integral part of `x` to a new variable `y` of type `int` using a technique called **type casting**. This converts data of one type into another type. The command `var_one = ( data_type ) var_two;` will convert `var_two` into the specified `data type` and store it in `var_one`.

A numerical example will help clarify this further: if the user enters `9.5`, which is a `float` type, the computer assigns `x = 9.5` as usual. We then type cast `x` into an integer and save it in `y`, which would make `y = 9` and the condition `y = x` or `9.5 = 9` fails, telling us that the user entered a non-integral quantity.

On the other hand if the user entered just `9`, then we would have `x = 9.0` and `y = 9` and

finally `x = y` making our condition `true` and telling us that the user did, in fact, enter an integer.

LINE 112: This `goto` statement inside the `if` section kicks into action when `y != x`, which, as we saw earlier, means that the user did not enter an integer. The `goto TRY;` command then tells the computer to go to the line labelled `TRY` and continue the program from there.

5.4. LOGICAL OPERATORS

One final, useful fact about `if` conditions is that multiple conditions can be combined together in the same statement. There are primarily two ways in which this can be done: either execute the specified set of commands if *at least one* of the given statements is true, or execute the specified set of commands only if *all* the given statements are true.

Such conditions are specified using one of three logical operators: the AND operator `&&`, the OR operator `||`, and the NOT operator `!`, all of which must be written in uppercase.

You have come across the logical NOT operator already in our definition of the 'not equal to' relational operator `!=` and the behaviour of the other two is well-known from mathematics and physics. For an AND operation to be true, all conditions must be satisfied; for an OR operation to be true, at least one condition should be satisfied.

In an `if` statement, each condition must be mentioned separately, parenthetically. The syntax `if((condition 1) && (condition 2) || (condition 3) && !(condition 4)) { }` is best understood using the example program below. In an attempt to stop hand-holding, no line-by-line explanations are given this time round.

```
121  #include<stdio.h>
122  int main() {
123      /* Check if a given number is an integer, a whole number or a
124         fraction, and also whether it is positive or negative. */
125
126      float x;
127      printf("\nEnter a number:\t");
128      scanf("%f",&x);
129
130      if ( x == 0 )
131          printf("\n%d is a whole number.\n",(int)x);
132      else if ( (x == (int)x) && (x > 0) )
133          printf("\n%d is a positive integer.\n",(int)x);
```

```
134         else if ( !(x != (int)x) && (x < 0) )
135             printf("\n%d is a negative integer.\n",(int)x);
136         else if (x > 0)
137             printf("\n%f is a positive fraction.\n",x);
138         else printf("\n%f is a negative fraction.\n",x);
139
140         return 0;
141     }
```

POSSIBLE ERRORS

Keep an eye on your brackets. They are most likely what will trip you when writing a program. When they can be avoided, prefer to avoid them. In the last example program we used as few brackets as we needed to. And, in the process, also try to write programs economically. Just like you would not leave superfluous constants lying around for no reason in your equations, do not declare more variables than you really need.

Compare the last two example programs and see how we accomplished type casting in the latest one without explicitly declaring a second variable. Lastly, whenever you write an `if` condition, ask yourself if the condition actually has a Boolean answer; if it does not, then your program will naturally fail: `if ( 3 - 2 ) { }` is *not* a sensible condition. Once you have written your program, try to read it like english and see if it makes sense — it should.

EXERCISE 2

1. The following algorithm is known as 'Euclid's algorithm' and is used to find the greatest common divisor of two integers:

   PRINT Enter two integers
   INPUT a and b
   ALPHA: **IF** $b = 0$ **THEN GOTO [GAMMA]**
   IF $a > b$ **THEN GOTO [BETA]**
   LET $b = b - a$
   GOTO ALPHA
   BETA: **LET** $a = a - b$
   GOTO [ALPHA]
   GAMMA: PRINT a

   Write a C program based on this algorithm.

2. Ask the user for five numbers, whether integers or fractions, and write a program that outputs the largest and the smallest of these numbers. If you need help getting started, try writing an algorithm first.

3. Accept three random lengths from the user and see if they form a triangle. (Use the fact that for a triangle of sides `x`, `y` and `z`, all three conditions, `x + y > z` and `y + z > x` and `z + x > y` must be satisfied.) If they do not form a triangle, ask the user for three new values.

***4.** Accept `a`, `b` and `c` for the quadratic equation $ax^2 + bx + c = 0$ and find its roots.

(Programs that are part of your coursework and exams are marked with an asterisk. Solutions to these programs are provided in section 10 of this handbook, but try solving them yourself first.)

TIPS

If you like, you can use the **ternary operator** shorthand for certain `if...else` statements that evaluate variables. Next time, test the syntax `condition ? expression1 : expression2 ;` and see where it works and where it fails. Do not forget to test nested statements. (Also see p. 34.)

6. FUNCTIONS AND LOOPS

6.1. FUNCTIONS SIMPLIFY PROGRAMS

In a program there may be certain lines of code that you use several times without alterations. You can always keep re-typing these lines over and over again or you can type them just once and keep referring back to them.

At first you may want to use the `goto` command for this, but the drawback with that approach is that `goto` continues the entire program from a particular line and has no way to specify a restriction on which lines to execute. Using functions solves this problem.

Another way to think of functions is like chunks of your program. If you write a program in several chunks, you can simply specify which chunk to execute in your main function and the computer will do the rest. Alternately, having functions lets you simplify complex programs: if something is wrong in a lengthy program it becomes hard to spot the error, but if we can narrow it down to a specific function spotting and correcting errors becomes much easier.

By now you are familiar with the `main()` function. The **function declaration** syntax is

```
data_type functionName( data_type var1, data_type var2 ){ ... return value; }
```

where the `functionName` has its first word conventionally written in lowercase and has successive words written with a leading uppercase letter. The arguments listed within parentheses must be **passed** to the function from elsewhere in the program *in the same order* and with the same data type, and the `return` value must match the data type of the function.

To better understand this idea, as always, let us write an example program. This time, let us make things interactive but keep the intent simple. Say we want to accept some numbers from the user and give them options of performing some basic arithmetic operations on them.

The program is long and looks complicated but is not. Think of yourself as a physicist reading a program now and try to understand it all by yourself. Use the many comments provided as an aid to help you out along the way. After the source code you can follow the brief explanation provided, that describes what a user will experience when they run this program.

```
142  #include<stdio.h>
143  #include<math.h>
144  #include<stdlib.h>
145
146  float a, b, option;              // Global variables
147  int counter=1;                   // declared outside for
148  char repeat;                     // use in all functions
149
150  int checkInteger( float given ) { // Variable called 'given' ,
151      if( given == (int)given )     // of type float, has
152          return (1);               // been declared for
153      else return(0);               // use only within
154  }                                 // this function
155
156  float acceptNumber() { // not all functions need arguments
157      float accept;
158      printf("\nEnter number %d:\t",counter);
159      scanf("%f",&accept);
160      return(accept); // returns the value of 'accept' (line |\ref{
             func_example_return_use}@*))
161  }
162
163  void printer( float result ) { // void functions do not return()
164      if(checkInteger(result) == 0)
165          printf("\n\nThe result is: %.3f.",result);
```

```
166         else printf("\n\nThe result is: %d.",(int)result);
167     }
168
169     int main() { // The program starts with the first line of main()
170
171         // Ask for an option
172      RE:system("clear");
173     OPT:printf("\n\nPlease pick an option [1-4]:\n");
174         printf("1. Add\t 2. Subtract\t 3. Multiply\t 4. Divide\n");
175         printf("5. Mod\t 6. Sine\t 7. Cosine\t 8. Tangent\n\n");
176         scanf("%f",&option);
177
178         // Check for valid option
179         if ( (checkInteger(option) == 0) || (1 > option) || (option >
                8) )  {
180             printf("\nInvalid option. Please pick again.\n");
181             goto OPT;
182         }   // with checkInteger(option) we send the value of
183             // 'option' to the checkInteger function above.
184
185         // Ask for as many numbers as required
186     TRY:if( (1 <= option) && (option <= 4) ) {
187             a=acceptNumber();
188             counter++;
189             b=acceptNumber();
190         }
191         else {
192             counter=1;          // Value returned by acceptNumber()
193             a=acceptNumber(); // function will be assigned to 'a'
194         }
195
196         // Check if a and b are integers
197         if ((checkInteger(a) == 0) || (checkInteger(b) == 0)) {
198             printf("\nIt looks like you have not entered an integer.\
                   nPlease try again.\n");
199             goto TRY;
```

```
200         }
201
202         // Perform operations
203         if (option == 1)
204             printer(a+b);
205         else if (option == 2)
206             printer(a-b);
207         else if (option == 3)
208             printer(a*b);
209         else if (option == 4)
210             printer(a/b);
211         else if (option == 5)
212             printer(fabs(a)); // math.h provides the fabs() function
213         else if (option == 6)
214             printer(sin((a*M_PI/180))); // trig functions in radian
215         else if (option == 7)
216             printer(cos((a*M_PI/180)));
217         else printer(tan((a*M_PI/180)));
218
219         // Another round?
220         printf("\n\nWant to try again (Y or N)?\t");
221         scanf(" %c", &repeat); //Try removing the space before %c
222         if ( (repeat == 'Y') || (repeat == 'y')) goto RE;
223         else exit(0);   // makes your C program quit automatically
224
225         return 0;
226     }
```

TIPS

Ironically, the quickest way to understand what various parts of a program do (especially the lines you are not clear about) is to delete or modify those lines in some way and see how it breaks the program.

Suppose that a user runs this program. They first thing that will happen is the global variable declaration (lines 146 to 146). Following this the computer starts off with the `main()` function as usual (line 169). Besides the `main()` function there are three other functions that we have

defined in this program: `checkInteger()` (line 150) which checks if a given number is an integer or not, `acceptNumber()` (line 156) which accepts inputs, and `printer()` (line 163) prints values.

The user is first provided a set eight of options (line 171) and they must pick one. We then check (line 178) to make sure they picked one of the eight and not something invalid. Following this, we ask them to either enter one or two numbers (line 185) depending on their option and then check if they entered integers (line 196, also see the exercise below). Once everything is set we perform the necessary arithmetic operations (line 202) and send the output to be printed, all with a single line of code. Note the `fabs()` function which outputs the absolute value of its argument; think of it as a shorthand used to replicate the behaviour of the command `if(x<0) x=x-(2*x);` for some `float` variable `x`.

Finally, we ask the user if they want to try everything again, i.e. start over and pick a new operation and new number/s. We use a character identified by `repeat` to check this: if the user assigns `repeat` a `Y` (or `y`) to say 'yes', then we return to that part of the program where we offer eight options, using `goto`. And if the user enters anything besides `Y` (or `y`) we take it as a 'no' and terminate the program. Lastly, line 223 is not compulsory. Using an `if` without an `else` will terminate the program as expected.

EXERCISE Observe that checking for integers in this program is not strictly necessary. It serves no purpose and the program should work even if the user enters fractions. Try to modify the program in such a way that entering fractions too is allowed. Do not forget to change the `printer()` accordingly.

6.2. LOOPING WITH `do` AND `while`

It may so happen that we come across a particular instruction that we want the computer to execute n times. A simple example of this would be adding n numbers. Commands used to execute repetitive instructions are called **loops**. The most basic looping command is the `while` loop.

For any loop to run, there must be some condition that is either valid for the entire lifetime of a loop and whose invalidity marks the end of a loop. For instance, you can *toss a coin* until *your thumb aches*: the act of tossing a coin is then the repetitive command you are performing and the condition that marks the end of the repetition is your aching thumb.

The syntaxes for the `while` and `do...while` loops make sense even from a grammatical perspective: `do { } while ( conditions ) ;` and this is the only instance where you will encounter conditions trailing a set of commands. As in the case of the decision making `if` statements discussed in section 5, all conditional statements henceforth will precede the list of

commands.

Before we discuss the exact manner in which the `do ... while` loop works, there is another, closely-related looping statement that demands our attention. The `while` statement (which is, arguably, the one used more often) has the syntax `while ( conditions ) { }` and primarily differs from the `do...while` block thanks to its condition appearing at the head of the block.

The key difference between these two loops is clear based on the order in which they are executed: the `do...while` loop first does something and then checks a condition and then repeats the same thing and checks the condition again and repeats so long as the condition to run the loop is satisfied. The `while` loop first checks the condition and runs the commands and checks the condition again and repeats so long as the condition to run the loop is satisfied. Beware of which loop you use, therefore: both function similarly except at the start, where the `do...while` loop executes the commands once regardless of whether the mentioned conditions are satisfied or not whereas the `while` loop gives the condition precedence. Understanding one of these loops with an example program will suffice (although the program, admittedly, does little). We shall prefer the simple `while` loop:

```
227  #include<stdio.h>
228
229  int main() {
230
231      /* Accept a limit and print that many natural numbers. */
232
233      int x=1, n=0;
234
235      printf("\nEnter a limit:\t");
236      scanf("%d",&n);
237
238      while ( x <= n ) {
239          printf("%d \t",x);
240          x++;
241      }
242
243      printf("\n");
244
245      return 0;
246  }
```

The output of this program is to simply print natural numbers from 1 to n. What difference would it make if the `do...while` loop were used instead? Specifically, think about what would happen if the user entered $n = 0$ with a `do...while` loop.

6.3. MORE ROBUST LOOPS WITH `for`

The `while` loops are somewhat basic in nature and one key property of loops can be incorporated into the loop syntax itself to make for simpler commands.

Observe the increment command in line 240 in our example `while` loop program. It so happens that such increments are a characteristic property of most loops: you will often want to perform a command once for a variable, then move to the next; or you may want to perform a command n times and use a counter that gets incremented with each round; or you may want to perform commands for one like of a table stored in a file and the repeat it for each successive line and so on. In all of these cases, the increment is an important factor.

Equally important, of course, is the condition, but notice the initialisation in the example `while` loop program. We initialised $x = 1$. (What would happen if we did not do that?) We will often want to initialise values in our loops, which means incorporating this process into the syntax rather than as a separate command would be economical. Indeed all of this is solved by the `for` loop: `for ( initialisation ; condition ; increment ) { }`.

Keep the order of the `for` arguments in mind: first initialise your variable, then state the condition which must be broken for the loop to be broken, and then state the values by which your counter variable must be incremented. The initialised variable and incremented variable need not be the same, i.e. you can choose to initialise the counter in advance, and this is often the route taken. Nonetheless, the `for` loop honours what variable you initialise, what condition you state, and what increments you define.

Here is the same program as the `while` loop example we previously saw, except it uses the `for` loop this time round:

```
247  #include<stdio.h>
248
249  int main() {
250
251      int x, n=0;
252      printf("\nEnter a limit:\t");
253      scanf("%d",&n);
254
255      for( x=1 ; x <= n ; x++ )
256          printf("%d \t",x);
```

```
257
258         printf("\n");
259         return 0;
260     }
```

The body of the `for` loop becomes smaller (small enough, in fact, that we have eliminated the braces enclosing the commands). Loops often find creative use in a program; whenever you have a bunch of tasks of the same nature to perform on a variable or a set of variables, prefer one of the three looping statements discussed so far.

EXERCISE 3

1. Accept two arbitrary limits and list the prime numbers between, and including, these limits.

   (Remember to handle all possible scenarios: what if the user gives the upper limit first and then the lower? Or what if the user gives a composite number as one of the limits? Make sure your program is complete and, to start with, determines the order of the user's given limits as well as the first and last prime numbers to be printed properly.)

2. Use loops to accept `a` and `b` and determine a^b without using the `pow` command or calling `math.h` at all. If you got this right, let us complicate the problem slightly: use functions and loops to write a program that extracts the ones, tens, hundreds etc. digits from a number.

   Hint: What does `x/((int)pow(10,n)) % 10` do for, say, $x = 326$ and $n = 1$, and why?

***3.** Find the roots of the quadratic equation $3x^3 - 9x - 5 = 0$ by bisection; start by asking the user for the two guesses characteristic of this method.

Use functions and a `while` loop. Observe how this construction helps by simplifying the next part of this problem: re-write the program for another equation.

***4.** Generate the first n elements of the Fibonacci[4] sequence. As a first case, do this using `while` looping, then re-write the program using `for` looping.

Finally, after you have warmed up with basic loops, try a second case: write the program so that you accept a random number from the user and generate a Fibonacci sequence of as many elements as necessary so that the greatest element in the sequence is as close as

[4] For Leonardo Fibonacci, pronounced 'fee-bone-*ah*-chee'.

possible to the number given by the user[5]

***5.** Approximate the roots of the equation $\frac{y+x}{yx}$ using the classical Runge-Kutta[6] method; start by asking the user for the two guesses characteristic of this method.

Use functions and a `while` loop. Observe how this construction helps by simplifying the next part of this problem: re-write the program for another equation.

***6.** Use the Trapezoidal tule to integrate $f(x) = x^4$. Request the user for the size of the interval (i.e. the accuracy of your approximation) and the limits of the integral.

TIPS AND POSSIBLE ERRORS

You can increment or decrement a variable `x` in three ways: the most straightforward method would be `x = x + 1` using the assignment and addition operators, but the problem with this is that the new `x` no longer refers to the old object `x` but to the newly calculated object which is then simply bound back to the old `x`; the second method is to use the shorthand `x++` that we have already come across, which only uses the addition operator and increments the object *in place*; and, finally, there is the form `x+=1` which works the same as `x++` but allows flexibility since we can choose to increment `x` by any n using `x+=n`. The same approaches work for subtraction too.

However, there are two ways to increment (or decrement) a variable based on when the variable is incremented. Try the following simple program that increments a number: `#include < stdio.h > int main() { int x = 7; printf("x = %d", x++); return 0; }` and you should get the output `x=7`. Next modify the program slightly to use `++x` instead: `#include < stdio.h > int main() { int x = 7; printf("x = %d", ++x); return 0; }` and you should get `x=8`. The difference between `x++` and `++x` is that, while the former executes the current statement and then increments `x`, the latter first increments `x` and then executes the current statement. Decide what your program logic needs carefully before picking one of these.

6.4. SWITCHING THINGS AROUND

Let us deviate from arrays briefly to look at an interesting type of decision making command. Recall the example program we discussed on p. 27. In it we used the `goto` command to offer the user some options and execute bits of code accordingly. There is a simpler way to do this using a `switch` statement. The syntax of a switch statement may be a little hard to grasp at one go, so we

[5]For example, in the first case, $n = 7$ must generate the sequence $0, 1, 1, 2, 3, 5, 8$ and, in the second case, the same $n = 7$ must generate the sequence $0, 1, 1, 2, 3, 5$.

[6]For Carl Tolmé Runge and Martin Wilhelm Kutta, pronounced 'roong' and '*koo*-tah' respectively.

will expand it in a moment. The syntax is intended to execute commands based on the user's choice: `switch ( var_1 ) { case var_2: statements; ... default: statements; }` The idea is to allow users to pick an option and, as a **fallback**, to execute some **default** commands:

```
261  switch ( choice_identifier ) {
262      case int_1 :
263          statements;
264      case 'X' :
265          statements;
266      default :
267          statements;
268  }
```

The switch statement associates the `choice_identifier` with the integral or character case identifier that the user chooses and then executes appropriate commands. Use integrals as usual or characters within single quotes, e.g. `'A'` or `'X'`. Lines 202 to 217 can be simplified using the `switch` statement as follows:

```
269  switch ( option ) {
270      case 1 :
271          printer(a+b);
272          break;
273      case 2 :
274          printer(a-b);
275          break;
276      default :
277          goto RE; // RE is the flag we set on line 172
278  }
```

After executing a case, without an explicit `break;` command, the program will move onto the next case. Almost always this is unwanted behaviour, so ensure you have the `break` command that tells the system to exit the `switch` statement after executing the required case. (Note that cases 3 to 8 have been omitted above since they should be clear from those shown in the example.)

Also recall the command `exit(0);` that takes the `break` command one step further and exits the program altogether. We came across this on p. 29.

7. ARRAYS AND STRINGS

7.1. HANDLING NUMERICAL DATA SETS

Often, in physics, particularly during simulations and experimentation, we end up with large data sets. The simplest case of this is a set of coördinates that can plot a function or help us determine the nature of the phenomenon under observation. When calculating with such data, it is not uncommon to have to perform the same calculations repeatedly; for a handful, perhaps manual repetition is not that cumbersome, but with sets of tens or hundreds of values or more, it helps to be able to compute and perform common operations over several variables in one go.

That is one case of handling data groups. The other classic case is that of a matrix, where all elements do not necessarily undergo the same process (e.g. matrix multiplication or row-column switching) but the entire group still has to be handled as a single entity.

It is hard to keep track of such individual variables, which is why C allows the use of an array or matrix of variables that can be treated as a single entity but operated on individually if we so choose. These are not special memory units; the onus is on the computer to keep track of the elements of an array rather than on the user.

We could call array elements as `a`, `b`, `c`... but this can clearly get complicated quickly. Instead we refer to them much like we do in physics or mathematics: with 'row' and 'column' numbers. You can also have three, four, five or higher dimensional arrays but the 'row' and 'column' analogy sadly breaks in the third dimension. Nonetheless, an array of integers called, say, `A` can be represented by `A[5]`.

There are a few things worth noting here:

— All elements of an array must be of the same data type

— Array counters start at zero and not one, so an array like `A[5]` has five elements, but from zero to four rather than from one to five: `A[0]`, `A[1]`, `A[2]`, `A[3]`, and `A[4]`.

The syntax for declaring an array is the same as that for declaring any individual variable of the same data type: `data_type name[size];`, so, for example, two one-dimensional integral arrays `P` and `Q` with two and four elements respectively may be declared as `int P[2], Q[4];` but their final elements are `P[1]` and `Q[3]`.

Initialising arrays is possible too: `float constants[3] = {3.14, 6.28, 0.707};` creates variables `constants[i]` with the corresponding values up to `i = 2`. Referring to the value stored in `constants[1]` later in your C program will then fetch the value of `6.28`. However, you do not have to initialise all values. For instance, `int charge[3] = {+1,-1};` will automatically set `charge[2] = 0`.

Further, simply declaring an array without explicitly mentioning its size is allowed so long as you initialise the array. In this case, the size of the array will be determined accordingly. Say you

declare `float squares[] = {4, 9, 16, 25, 36};` then the processor automatically sets the size of `squares[]` to `5`. Remember to consciously count elements from zero since this rarely comes naturally to most students in the beginning.

Finally, two-dimensional arrays are created by specifying the sizes of the two dimensions separately: `data_type name[size1][size2];` The initialisation of two- and multi-dimensional arrays is as follows: `float identity[3][3] = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };`

As before, we attempt to understand basic array operations using an example program. Our intention is to see how array inputs can be accepted and how an array can be printed. Since we will be using matrices, let us also try to make our matrix output look like a matrix and not just a set of numbers.

For simplicity, let us see how two 3×3 matrices may be added. You will later have to follow closely and extend this idea to multiplying matrices and more.

```
279  #include<stdio.h>
280
281  int a[3][3],b[3][3],c[3][3],row,column;
282
283  void acceptor(int matrix[3][3]) {
284      for( row=0 ; row<3 ; row++ )
285          for( column=0 ; column<3 ; column++ )
286              scanf("%d",&matrix[row][column]);
287  }
288
289  int main() {
290  // Accept elements of matrix A
291      printf("\n\nEnter the elements of matrix A by row:\n");
292      acceptor(a);
293  // Accept elements of matrix B
294      printf("\n\nEnter the elements of matrix B by row:\n");
295      acceptor(b);
296  // Add matrices A and B as C
297      for( row=0 ; row<3 ; row++ )
298          for( column=0 ; column<3 ; column++ )
299              c[row][column] = a[row][column] + b[row][column];
```

```
300   // Printing matrix C
301       printf("\n\nThe sum of the two matrices is:\n");
302       for( row=0 ; row<3 ; row++ ) {
303           for( column=0 ; column<3 ; column++ )
304               printf("%d\t",c[row][column]);
305           printf("\n");
306       }
307       return 0;
308   }
```

By now you should be able to follow this simple program without trouble. To aid you, here are a few things to observe:

1. We use global variables to prevent redefining `row` and `column` in our `acceptor()` function.
2. In our `acceptor()` function we have the preset size of 3×3. Can you pass an array to a function without specifying its size?
3. The `acceptor()` function is `void` since it does not return any value to the `main()` function, rather it assigns values to the elements of the given arrays.
4. We could have used functions to add and print too, but it serves no purpose since we only add and print once. However, should you choose to write functions for them, it will only vary slightly from the `acceptor()` function in the example program.
5. Remember to always `return 0` by the end of the `main()` function.

7.2. HANDLING WORDS AND SENTENCES

We will not concern ourselves too much with string handling for the purposes of this course, but it is nonetheless important that we have some basic ideas about it. A **string** (or a word or sentence as we know it better) is simply treated as an array of characters in C. Whenever you want to use strings, be sure to `#include<string.h>` as a preprocessor.

The word 'Physics', according to us, carries seven letters, but, according to the processor, carries eight. The **null character** `\0` marks the end of a string is always appended to it. The word 'Physics', then, is a character array defined by `char subject[8] = "Physics";` and spaces too, if any, must be counted as characters. Alternately, we can also define the same string using `char subject[8] = {'P', 'h', 'y', 's', 'i', 'c', 's', '\0'}` but, in this case, we will have to specify the null character explicitly.

The term **string handling** encompasses a lot of potential operations one can perform with strings, somewhat (but not entirely) analogous to the operations we discussed in section 5. String handling functions include the *concatenation* (combining) of two strings using the `strcat`

command: `strcat("string1","string2");`, calculating the length of a string using the `strlen` command: `int_var = strlen("string");`, reversing a string using the `strrev` command: `strrev("string");`, copying strings from a `source_identifier` to a `destination_identifier` using the `strcpy` command: `strcpy(destination_identifier,source_identifier);`, and, finally, comparing the ascii difference between a pair of strings using the `strcmp` command: `int_var = strcmp("string1","string2");`.

String comparison works the same as character comparison which we discussed briefly in section 4.4, i.e. it compares the ascii values of each letter of the two strings based on the idea that (if the strings are, in fact, the same) the difference between two corresponding letters will always be zero. This is why the output of `strcmp` must be assigned to some integral variable `int_var` using the assignment operator as shown in the syntax above. The same reasoning works for the assignment performed in the `strlen` syntax as well.

As it turns out, the two other important functions we normally perform with strings, accepting and displaying them, are both easier than performing the same operations on numerical arrays.

Accepting strings with `scanf` is likely the easiest method `scanf ( "%s" , identifier ) ;` but has a crippling problem. Recall how you would conveniently enter two numerical inputs as, say, `7 8` in your previous programs. The `scanf` statement would treat the space between your two inputs as a separator and call the input before the space as the first one and the input after the space as the second.

The same logic, unfortunately, is extended to strings as well and breaks down for obvious reasons: whereas `scanf` for the input `Physics` works great, the input `Condensed matter physics` sees only `Condensed` being accepted and the `scanf` statement ends as soon as it encounters a space. This (mis)behaviour can be worked around using `scanf(" %[^\n]",identifier);` which accepts all inputs until the first new line `\n` character. (Note the space before %.)

The most convenient method of accepting (or *getting*) a string from the user is with the `gets` function: `gets ( identifier ) ;` accepts and stores a string from the user up to the new line character (i.e. until the user hits the Enter/Return key on their keyboard), mimicking the behaviour of `%[^\n]` but with much more elegant code.

Finally, we are left with printing the strings we have in memory. This is done with the usual `printf` command, this time without much trouble: `printf ( "%s", identifier ) ;` does it. Here is an example program where we ask the user for the multiple or submultiple of units and return the corresponding scaling factor.

```
309  #include<stdio.h>
310  #include<string.h> // For strings
```

```
311  #include<stdlib.h> // For exiting
312
313  int main() {
314
315      // Defining unit as an array of strings
316      const char *unit[19]; // Pointer called unit
317      unit[0] = "nano"; unit[1]="\0"; unit[2]="\0";
318      unit[3] = "micro"; unit[4]="\0"; unit[5]="\0";
319      unit[6] = "milli"; unit[7] = "centi"; unit[8] = "deci";
320      unit[9] = "the unit itself"; unit[10] = "deka";
321      unit[11] = "hecto"; unit[12] = "kilo"; unit[13]="\0";
322      unit[14]="\0"; unit[15] = "mega"; unit[16]="\0";
323      unit[17]="\0"; unit[18] = "giga";
324
325      // Defining other variables
326      char old_unit[20], new_unit[25];
327      char option[10], prefix[20];
328      int i;
329      printf("\nDo you know the (a) order or (b) prefix?\t");
330      scanf(" %[^\n]",option);
331
332      switch (option[0]) {
333        case 'a':case 'A':case 'o':
334          printf("\nEnter power (-9 to +9):\t");
335          scanf("%d",&i);
336          i=i+9;
337          if( (0 <= i ) && ( i <= 18)) { // Within bounds of *unit
338              if( (strcmp(unit[i],unit[1])) ) { // Exists in *unit
339                  printf("\nEnter unit:\t");
340                  scanf(" %[^\n]",old_unit);
341                  strcpy(new_unit,unit[i]); // Copy and append as
                         we
342                  strcat(new_unit,old_unit); // cannot append to
                         array
343                  printf("\nThe new unit is %1s.\n\n",new_unit);
344              }
```

```
345                 else {
346                     printf("\nSomething seems to be wrong.\n");
347                     printf("(Enter powers as multiples of three.)\n\n
                           ");
348                     exit(0);  // In case of error in second if
349                 }
350             }
351             else {
352                 printf("\nSomething seems to be wrong.\n");
353                 printf("(Enter powers between -9 and +9.)\n\n");
354                 exit(0); // In case of error in first if
355             }
356             break;
357 
358         case 'b':case 'B':case 'p':
359                 printf("\nEnter the prefix:\t");
360                 //fgets(prefix,100,stdin);
361                 scanf(" %[^\n]",prefix);
362                 for(i=0; i<=18; i++) {
363                     if(!(strcmp(prefix,unit[i]))) {
364                         i = i-9;
365                         printf("\nThe order is %d.\n\n",i);
366                         break; // Without break, for loop runs again
367                     }
368                 }
369                 break;
370     }
371 }
```

You may not fully understand what line 316 is doing just yet but you should be able to make an educated guess. Observe certain other lines, though, where we accept strings, output strings, and even compare them. Particularly note how we use `if(!(strcmp()))` on line 363 to compare strings and get a Boolean answer regarding their equivalence.

7.3. BUILDING LARGER PROGRAMS

We previously declared global variables whenever we wanted to use them between functions since any variable declared inside a function can be accessed only within that function.

The trouble with global variables is obvious: you may redeclare a variable with the same name elsewhere in your program (especially as your program gets longer) and end up overwriting it inadvertently, or you may lose track of how and where you have manipulated a global variable. On the processor side, looking up global variables takes longer than local ones[7]. Global variables can also be a security risk as they can be accessed by other programs using the `extern` keyword.

When we want to cross-reference variables across functions, we can use **static variables** as a fairly good alternative to global variables. They differ from global variables in some important ways: firstly, static variables are only initialised once, at the start of the program; secondly, like a global variable, we can declare static global variables too; thirdly, unlike local variable, a static local variable can be accessed by other functions so long as one of the two functions cross-referencing it is the function inside which the variable is declared.

Declaring and initialising static variables is done in the same manner as any other variable but with the `static` keyword prepended: `static int x = 7;` creates a variable identified by `x` with the initial value `7` that can be manipulated by any function in conjunction with the parent function of `x` and its value can be modified in any manner but never reset or reinitialised.

Execute the example program below to understand static variables. Try to reason out what the output of the program will be before executing it.

```
372  #include <stdio.h>
373  void parent() {
374      int i=0;              // i is just another int variable
375      static int j=0;       // j is a static variable of type int
376      printf("i = %d\tj = %d\n", i++, j++);
377      return;               // Function of type void returns nothing
378  }
379  int main() {
380      for(int n=-1;n<5;n++)
381          parent();
382      return 0;
383  }
```

[7] This is only a matter of tiny fractions of a second, but it can add up on longer programs.

Notice that the static variable `j` remains untouched. In effect, you can use static variables if you want to remember their values across functions, sort of like memory storage variables. Here are ideas for further exploration: What happens if you declare `static int j` globally? What, if any, is the difference between a global static variable and a global regular variable if you declare another variable `int j` only inside your `main()` function? This next paragraph may help you answer better, so return to these questions later.

Like a static variable, you can declare a static function too: `static data_type name(){}` creates a function that can only be used inside its own `c` file. By definition, all functions are global and can be used across c files. Let us look at an example of how this works for variables and functions. The following directory tree is assumed:

```
Calculator ..... Main directory
├── variables.h ..... Header file (extern)
├── calc.c ..... c file (arithmetic)
├── operator.c ..... c file (organisational)
└── main.c ..... c file (main function)
```

Like any other example program in this book, to understand this program, it would help if you typed these files as well as compiled and executed `main.c` and followed the explanations given below. To start off, note that the four files listed above have the following contents:

```
<< variables.h >>
384  extern int a, b, choice, x;
385  extern void input();
386  extern void selection();
387  extern int add();
388  extern int mul();
389  extern void output();

<< calc.c >>
390  #include <stdio.h>
391  #include "variables.h"
392  int x;
393  int add() {
394      x=a+b;
395      return (x);
396  }
397  int mul() {
398      x=a*b;
399      return (x);
400  }

<< operator.c >>
401  #include <stdio.h>
402  #include <stdlib.h>
403  #include "variables.h"
404  int a, b, choice;
405  void input() {
406  printf("\nEnter two nos: ");
407  scanf("%d %d",&a,&b);
408  return;
```

```
409   }
```

```
410   void selection() {
411   printf("\n\nPick one:\n");
412   printf("1. Add \t\t 2.
          Multiply\n");
413   scanf(" %d",&choice);
414   switch (choice) {
415      case 1:
416          output(add());
417          break;
418      case 2:
419          output(mul());
420          break;
421      default:
422          printf("\n Error.\n");
423          exit (0);
424      }
425   return;
426   }
427   void output(x) {
428       printf("\n\n%d\n\n",x);
429       return;
430   }
```

<< main.c >>

```
431   #include <stdio.h>
432   #include "variables.h"
433   int main() {
434       input();
435       selection();
436       return 0;
437   }
```

These files are quite interesting even if their working may not be straightforward at first glance. First of all, observe how we included our own header file. We can include any number of `c` files *without* a `main()` function inside a `c` file *with* a `main()` function this way too. This involves the `#include` command as we discussed way back on p. 4, but within inverted commas rather

than angle brackets since we are referring to a file we created *within the same directory* as the file with the `main()` function. When running, ensure you execute the file with the `main()` function since your program starts with it. Specifically, once you have navigated to the `Calculator` directory, the following commands in the terminal should compile it for you:

```
$ gcc -Wall main.c operator.c calc.c -o exe && ./exe
```

The `-Wall` option provided by `gcc` turns on some useful warning flags that, if your program is erroneous, should help you debug it. Of course, the command will work without `-Wall` too; and you can name the executable `exe` file whatever you wish. You can also compile each `.c` file separately, creating machine-readable **object files** (`.o`) that contain compilation-level output data:

```
$ gcc -c calc.c -o calc.o
$ gcc -c operator.c -o operator.o
$ gcc -c main.c -o main.o
$ gcc -c calc.c -o calc.o
$ gcc calc.o operator.o main.o -o exe && ./exe
```

This way (and especially if you include `-Wall` now) you will be able to pinpoint the source file/s with any error/s. But let us understand how such an arrangement of files would work harmoniously. It should give you a better idea of variables and functions on a local and global scale. Particularly, we will not be using `static` here (try it: the compiler will throw an error). We will be focusing on fetching variables and using functions across `c` files using `extern` instead.

1. The `variables.h` file coördinates a list of variables and functions defined in other files. The use of the `extern` keyword makes a promise to the compiler that said variable or function is defined elsewhere and need not be created.
2. The actual variable may be referenced by any number of different files (e.g. the variable `a` is used in `operator.c` and `calc.c`) but it must only be defined in one file to prevent conflicts (e.g. `a`, `b` and `choice` are defined only by `operator.c` like usual variables, and, likewise, `x` is defined only by `calc.c`). In fact, our `main.c` file with the `main()` function is the smallest file and simply calls other functions defined in other files.
3. The usage is as follows:
   — `main.c` calls `input()` which, `variables.h` tell us, is in `operator.c`
   — `input()` accepts two numbers
   — `main.c` calls `selection()` which, `variables.h` file tells us again, is in `operator.c`

— `selection()` calls `add()` and `mul()` in turn from `calc.c`, operates and displays results
— `main.c` signals that the program has ended

It must be emphasised that the `extern` keyword does *not* create a memory location for the variable; instead, it tells the compiler that such a variable (or, indeed, function) exists elsewhere. The variable itself must be compiled elsewhere and strictly only once.

The usefulness of `extern` and custom header files becomes apparent when you realise that the next time you want to accept two variables, you can simply turn to the `variables.h` and `operator.c` files rather than writing the same code again. By taking things a beyond just function blocks within a single file, we have created a simple yet effective method to write extremely large programs with clean, reusable code that is easy to understand and debug. To safely limit the **scope** of a variable to a single file, remember to declare it as `static`.

EXERCISE 4

1. This is a lighthearted program to appreciate the use of arrays for purposes other than in a matrix. Store the mass, charge and spin properties (and any others you can think of) of electrons, protons and neutrons (and any other favourite particles you may have) in individual *global* arrays and write a program that asks the user up to three questions about the properties of these particles and, based on the inputs, guesses which particle it may be.

2. Use arrays, functions and any other capabilities of a C program you have learnt so far to improve your earlier program coded for problem 2 in exercise 2 to determine the smallest and largest of five numbers provided by the user. You should now be able to make it considerably simpler than last time.

   Once that is done, improve your program so that, in addition to what it does now, it also displays the five numbers in ascending order and in descending order. Make sure your program works under these **use cases**: the numbers are already input in ascending or descending order; two or a few of the numbers are equal; the numbers are all equal.

3. Use arrays and functions to ask the user the elements of a 3×2 matrix `A` and a 2×3 matrix `B`, then prove that `A`·`B` $\neq$ `B`·`A`. Remember to display any results/outputs as matrices.

   Next, use the global functions from this program to write another program that computes the determinant of a matrix that the user provides.

4. You were instructed to declare global variables in program 1. Now use those to write a simple program that calculates the specific charge of your choice of elementary particles.

Make any changes necessary, use `extern`, and ensure your program is descriptive and makes complete sense to the user. In fact, this last point is something you should keep in mind for *any* program you ever write.

The next six programs are general-purpose and not necessarily restricted to section 7.

5. Compute the $\mathtt{n}^{\text{th}}$ root of a number given by the user. Let the user choose `n` too.

   HINT The `pow` operation works, but will using an integral expression such as `1/3` work? Recall that `1/3` evaluated as an integer is not `0.333` but simply `0`. Use `float` expressions like `1.0/3.0` instead.

6. Add $x^{-3}\ \forall\ 1 \leq x \leq 10\ ;\ x \in \mathbb{Z}$ and display the result. If you are confident you can do this, start off with this modification: compute the same function for $1 \leq x \leq n\ ;\ x \in \mathbb{Z}$ in steps of m, where the user gives you both n and m.

7. Produce a neat table of the three basic trigonometric functions for $0°$, $30°$, $45°$, $60°$, $90°$.

8. Add, subtract, scalar multiply, and check for orthogonality, parallelism or obliqueness, two three-dimensional vectors given by the user.

9. The Maclaurin series of e^x is given by

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

   Compute this up to n terms as the user desires. You are *not* allowed to use the `math.h` function `exp(identifier)` anywhere in your program. (Use it only to verify your result.)

10. Complete the calculator program from our example above (lines 384–437). Make sure it performs all the basic operations of a scientific calculator and any more you can think of.

8. FILE HANDLING

While a C program can be self-sufficient the convenience of automating using a program is best seen when the program can read and write to other files. We saw quite a specific example of this in 7.3 but our communication there was between other program-oriented files. What if we have a regular text file or a data file output from another program or experiment?

The purpose of file handling is to allow a C program to read from or write to another file. Files of most generic types have reasonable support here (for example, it is still foolish to want to edit a pdf file) and file permissions remain as important as elsewhere in the system. However, before we discuss how files are handled we will have to take a moment to understand the notion

of **pointers**, a special type of variable used in C and several other languages, which can find powerful use in several instances. (Following this should also help you fully appreciate line 316 that you had probably guessed your way through earlier.)

8.1. POINTERS

Test

* EXERCISE Enter a list of numbers into an `input.txt` file, one on each row. Then write a program to read that data from the file (ask the user for the file name and number of rows of data) and output the sum of those numbers and the average.

9. FURTHER READING

10. SOLUTIONS TO COURSEWORK PROGRAMS

10.1. FINDING THE ROOTS OF A QUADRATIC EQUATION

```
1   #include<stdio.h>
2   #include<math.h>
3   #include<stdlib.h>
4
5   int main() {
6
7       float a,b,c;
8       float d,root1,root2;
9
10      system("clear");
11
12      printf("\n\nEnter a, b and c of quadratic equation ax^2 + bx
            + c below: \n\n");
13      scanf("%f%f%f",&a,&b,&c);
14
15      d = b * b - 4 * a * c;
16
17      if(d < 0) {
18          printf("\n\nThe equation has complex roots: ");
19          printf("%.3f%+.3fi",-b/(2*a),sqrt(-d)/(2*a));
20          printf("and %.3f%+.3fi.\n\n",-b/(2*a),-sqrt(-d)/(2*a));
21
22          return 0;
23      }
24      else if(d==0) {
25          printf("The equation has two equal roots.\n\n");
26          root1 = -b /(2* a);
27          printf("They are both %.3f.\n\n",root1);
28
29          return 0;
30      }
31      else {
32          printf("The equation has are real roots.\n\n");
```

```
33            root1 = ( -b + sqrt(d)) / (2* a);
34            root2 = ( -b - sqrt(d)) / (2* a);
35            printf("They are %.3f and %.3f.\n\n",root1,root2);
36
37            return 0;
38        }
39    }
```

10.2. FINDING THE ROOTS OF AN EQUATION BY BISECTION

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #define error 0.0001
5
6   double functionOf(double x) {
7       // Code in the required function below
8       return (3*x*x*x - 9*x - 5); // 1 and 2 are good guesses here
9   }
10
11  int main()
12  {
13      double guess_one,guess_two,guess,solution,soln_one,soln_two,
            previous;
14      int root_exists=0,root_found=0,i=0,decor;
15      double f(double);
16
17      system("clear");
18
19      printf("\n\nThe equation being used is 3x^(3) - 9x - 5 = 0\n\
            n");
20      printf("\nEnter two guesses:\t ");
21
22      while(root_exists==0) // Check guesses
23      {
24          scanf("%lf %lf",&guess_one,&guess_two);
25          soln_one=functionOf(guess_one);
26          soln_two=functionOf(guess_two);
27          if(soln_one*soln_two>0)
28          {
29              root_exists=0;
30              printf("\n\nThe root does not lie between %lf and %lf
                    .\n",guess_one,guess_two);
31              printf("\nPlease enter two new guesses:\t ");
```

```
32          }
33          else
34              root_exists=1;
35      }
36      printf("\n\nThere is a real root which lies between %lf and %
           lf.\n\n",guess_one,guess_two);
37      while(root_found==0) // Solve till a root is found
38      {
39          printf("\nIteration %d\n",i);
40          for(decor=0; decor<=12; decor++)
41              printf("=");
42          printf("\n\na[%d](-ve) \tb[%d](+ve) \tx[%d] \t\tf(x[%d])\
               n",i,i,i+1,i+1);
43          printf("%lf\t",guess_one);
44          printf("%lf\t",guess_two);
45          guess=(guess_one+guess_two)/2;
46          solution=functionOf(guess);
47          printf("%lf\t",guess);
48          printf("%lf\n\n",solution);
49          if(solution < 0)
50              guess_one=guess;
51          else
52              guess_two=guess;
53          if(fabs(previous-solution) < error)
54              root_found=1;
55          else
56              previous=solution;
57          i=i+1;
58      }
59      printf("\n\nThe root of f(x) is %lf to an accuracy of %f\n\n"
           ,guess,error);
60
61      return 0;
62  }
```

10.3. GENERATING THE FIRST N ELEMENTS OF THE FIBONACCI SEQUENCE

```
#include<stdio.h>

int main() {

    int a=0, b=1, c, n, counter=0;

    printf("\nEnter a limit:\t");
    scanf("%d",&n);
    n-=2;
    printf("%d \t",a);

    do {
        printf("%d \t",b);
        c=a+b;
        a=b;
        b=c;
        counter++;
    } while ( counter <= n ) ;

    printf("\n");

    return 0;
}
```

10.4. APPROXIMATING THE ROOTS OF AN ORDINARY DIFFERENTIAL EQUATION USING RK4

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

float functionOf(float x, float y){
    // Code in the required function below
    return (y+x)/(y*x);
}

int main(){

    float K, K1, K2, K3, K4;
    float x0 , y0, x, y, i, interval;
    int j, n;

    system("clear");

    printf("\n\nSuggest any initial value for x:\t");
    scanf("%f", &x0);
    printf("\n\nSuggest any initial value for y:\t");
    scanf("%f", &y0);
    printf("\n\nEnter the number of iterations needed:\t");
    scanf("%d", &n);
    printf("\n\nEnter the skip between interations:\t");
    scanf("%f", &interval);

    printf("\n\n");

    x = x0;
    y = y0;
    for(i = x+interval, j = 0; j < n; i += interval, j++){
        K1 = interval * functionOf(x , y);
        K2 = interval * functionOf(x+interval/2, y+K1/2);
        K3 = interval * functionOf(x+interval/2, y+K2/2);
```

```
35            K4 = interval * functionOf(x+interval, y+K3);
36            K = (K1 + 2*K2 + 2*K3 + K4)/6;
37            x = i;
38            y = y + K;
39
40            printf("\t\t\t x = %.2f \t y = %.4f\n", x, y);
41
42            // NaN is an error about which we need provide an
                 explanation to the user (also, inf means infinity)
43            if( isnan(x) || isnan(y) )
44                printf("\nThere seems to be an error in line %d. \
                     nCheck your suggested initial values \nsince not
                     all initial values work \nfor all functions.",j);
45        }
46
47        printf("\n\n");
48
49        return 0;
50    }
```

10.5. APPROXIMATING DEFINITE INTEGRALS USING THE TRAPEZOIDAL RULE

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

float functionOf(float x){
    // Code in the required function below
    return(pow(x,4));
}

int main()
{
    int i,n;
    float x0,xn,h,t,y[20],so,se,ans,x[20];

    system("clear");

    printf("\n\nEnter the two limits:\t");
    scanf("%f%f",&x0,&xn);
    printf("\n\nEnter the width of an interval:\t");
    scanf("%f",&h);

    if(xn<x0){
        t=xn;
        xn=x0;
        x0=t;
    }

    n=(xn-x0)/h;
    if(n%2==1)
    {
        n=n+1;
    }
    h=(xn-x0)/n;

```

```
35        system("clear");
36
37        printf("\n\n\nSummary:\nUpper limit = %f\nLower limit = %f\
             nNumber of intervals = %d\nWidth of an interval ~ %f",xn,
             x0,n,h);
38        printf("\n\nThe values of y are\n");
39        for(i=0; i<=n; i++)
40        {
41            x[i]=x0+i*h;
42            y[i]=functionOf(x[i]);
43            printf("\t\t\ty%d = %f\n",i,y[i]);
44        }
45        so=0;
46        se=0;
47        for(i=1; i<n; i++)
48        {
49            if(i%2==1)
50            {
51                so=so+y[i];
52            }
53            else
54            {
55                se=se+y[i];
56            }
57        }
58        ans=h/3*(y[0]+y[n]+4*so+2*se);
59        printf("\nThe integration yields %f\n\n",ans);
60
61        return 0;
62    }
```

10.6. CALCULATING THE SUM AND AVERAGE OF DATA STORED IN A FILE

```
63  #include<stdio.h>
64
65  int main()
66  {
67      FILE *input;
68      int n,i;
69      float x[20],sum=0,avg;;
70      char name[20];
71
72      printf("\n\nEnter the name of the input file: \t");
73      scanf("%s", name);
74
75      printf("\n\nEnter the number of rows of data:\t");
76      scanf("%d",&n);
77
78      input = fopen(name,"r");
79
80      for(i=0; i<n; i++){
81          fscanf(input,"%f",&x[i]);
82          sum=sum +x[i];
83      }
84
85      fclose(input);
86
87      printf("\n\nThe sum is %3.2f.",sum);
88      printf("\n\nThe average is %3.2f.\n\n",sum/i);
89
90      return 0;
91  }
```