Computer applications for M.Sc-2

# Contents

# Chapter 1

# An Oversimplified introduction to C

## 1.1 Introduction

`C` is a medium level language. The more a language is like English, the higher is its level. It then tries to mimic the thought processes of the human mind. The lowest level language is machine language or, almost at the same level, assembly language. `C` is somewhere in between. Thus we get the power and control of assembly language along with much of the ease of programming of the higher level languages like `FORTRAN`, `Pascal`, `BASIC`, etc.

## 1.2 Compilation

High level code needs to be compiled. A compiler is a programme that can translate a programme written in a high level language into something the computer can run. Both the source code (the `C` programme) and the executable (the compiled code) exist as files in the hard disk of the computer and have different names. The source code is written by the user and has an extension of `.c` e.g. `bhutum.c` Here the `.c` is in small letters. After compilation, the compiler creates the executable file. It is not written by the user, but the user can name it at will. The compilation command is:

```
gcc bhutum.c -o bhutum
```

Here, the name following the `-o` option is the name that the compiler will give to the executable program file. Here the name of the executable is `bhutum`.

Now, there are two files in the hard disk, named `bhutum.c` and `bhutum`. [1]If you use mathematical functions like `sin`, `cos`, etc., [2]the compilation command will change to

```
gcc bhutum.c -o bhutum -lm
```

Here you are asking the compiler to link the mathematical library also.

To run the programme, you have to ask the computer to execute the executable file as a command, thus:

```
./bhutum
```

Here the `./` refers to the fact that the programme needs to be loaded from the current directory. [3]

## 1.3 Simple C

### 1.3.1 The Preprocessor

When a `C` programme is compiled, there is a *preprocessing* pass. Lines in the source code beginning with `#` are interpreted by the preprocessor. What the preprocessor does is to make changes in the source code itself before the actual compilation can proceed. For instance,
`#include <gablu>`
will cause the file `gablu` from the `/usr/include` directory to be written in place of this line. Similarly,
`#define PR printf`
will replace all occurrances of `PR` in the file with `printf`. The preprocessor is actually much more sophisticated than this. It can handle logic and conditional operations. If you want to see what the preprocessor does, just use the `-E` flag during compilation. Thus:

[1]Remember to give a name different from the name of the source file .. if you type `gcc bhutum.c -o bhutum.c` then the original source file will be deleted - a common mistake!

[2]The `math.h` header has to be included in this case, see below.

[3]This can be made automatic by modifiying the `PATH` environmental variable in the `.bashrc` file. Somewhere near the end you have to put two lines:
`PATH=$PATH:.`
`export PATH`
On logging in next time, (or closing the shell and opening a new one) the new settings will take effect. The directives tell the shell to add the current directory, represented by the dot, to the end of the executable search path. Now you can do away with the `./`.

`gcc -E a.c -o a1.c`
Will generate a source file `a1.c` from the file `a.c`, but all the preprocessor directives will have been executed to the full.

### 1.3.2 Headers

Most `C` programmes begin with some preprocessor directives which include special files, called *headers* usually from the `/usr/include` directory at the top of the programme. These are needed to help the compiler to know some details about the functions used. For instance, in order to handle standard input and output, the header file `stdio.h` has to be used. Similarly, `string.h` is used for string handling, `math.h` for relatively complex mathematics like `sin`, `cos` and such functions and `stdlib.h` is used for things like random numbers and many complicated functions.

The way to use headers is to include them, generally at the beginning of the source code, like this:

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

The <> delimiters mean that the files have to be looked for in what is called an *include path*, generally `/usr/include`, `/usr/X11/include` and some other standard libraries. There are ways to change the include path,[4] but this is not generally required by us. If we want to include files from the local (current) directory, say a file name `gablu.h`, then we will use the directive:

```
#include "gablu.h"
```

Often, large portions of commonly ocurring code is written into separate header files and included as needed in the final programme. [5]

### 1.3.3 The basic structure of a C programme

A `C` programme has headers at the top, [6]and *whitespaces* wherever you like. A *whitespace* is any blank spaces including blank lines that you may include

[4] A `-I` option in the compilation line does this, like `gcc -Ihalum bhutum.c -o bhutum` will temporarily expand the include path to look for files in the `halum` subdirectory under the current directory.

[5] This is of particular use in `C++` programmes where the classes are usually defined inside header files.

[6] you may leave out the headers, but then you will have difficulty in using many functions

to improve readability. *Whitespaces* are ignored during compilation and have no bearing on the final machine code.[7]

Beneath the `include` directives, the programme is supplied with a name which is `main` if the programme has to start from that point. The programme body is next enclosed between `{}` [8]and a few instances of `return` may be inside the body, notably at the end.

A Typical C programme now looks something like this:

```
#include <stdio.h>

int main(void)
{
.........
.........
return(1);
.........
.........
return(0);
}
```

The $\cdots$ represent lines of code.

#### 1.3.3.1 The meaning of `int` and `return` in this context

We will later learn about *subroutines*. They are also called *functions* here. *Functions* are programmes that are run by other programmes. Now, for us the programme that we write is all we are concerned about. However, for the computer, the main programme that is always running is the *kernel* or the main part of the *Operating System (OS)*. For it, our main programme is also a *function*. It is named `main` so that the *OS* knows that that is where one has to start to execute the user's code.

The `main` subroutine can take some *arguments* inside the first brackets. By writing `void` there (or leaving it blank) we tell this programme that we are not interested in any values or arguments that the *OS* may supply to the main programme.[9]

[7] In certain cases, you may find that leaving a blank line somewhere causes a change in the result! If such a thing happens, be sure that there are memory allocation problems in you programme, i.e. probably the variables have not been declared or used properly.

[8] A `C` programme is supposed to be a single line! Since a single line actually does very little, we use `{}` to join many lines into a single complex line!

[9] The usual form of the arguments is `(int argc, char **argv)`. This is used to access *command-line parameters* from the programme.

The `int` in front of main tells the *OS* that the programme will return some integer values on completion. We do not use these values here and could have used the other syntax `void main(void)` to do away with the `return` statements. However, the `return` statement is an easy way to allow the programme to exit from within different places of the code depending on different situations. This is an useful functionality and we will generally use this form. What value it returns is immaterial in the present context.[10]

Each normal executable line of a `C` programme has a semicolon `;` at the end. In this way, large lines of code may extend over many lines of text but still be considered as the same line. [11]

*Comments* are another useful feature. Any string beginning with `/*` and ending with `*/` is considered a comment. It is simply ignored during compilation and is there to help us remember what we have done or others to understand our code. Profuse use of comments is a highly desirable practice.

### 1.3.4 Variables in C

A variable is a region in memory that holds a certain value, analogous to our use of $x$ and $y$ in algebra. Variables are *declared* at the beginning of a programme, just after the `{` and before any executable statement is written.

#### 1.3.4.1 Names of variables

In order for us to be able to use a large number of variables, multi-character names are allowed, unlike the simple $x$ or $y$ that we use in algebra. Thus

```
this_is_the_variable_that_holds_the_value_of_the_ransom_money
```

is a completely valid name for a variable![12] Do not use the `+`, `-`, `*`, `/` signs inside the names (the *underscore* `_` is allowed) as also blanks, tabs, punctuation marks and other weird things. You can mix capital letters and small

[10] Actually, different values of `return` from the main could be used to signal to the *OS* different conditions at exit. Thus we could decide to use some value if the programme has exited due to the risk of a division by zero. Any condition we like could be flagged by the programme using different return values.

[11] This may be contrasted with a language like `FORTRAN` (`FORTRAN 4` or `77` - not `90` or `95`) where lines end at the $72^{nd}$ column or as soon as a linebreak is encountered, whichever comes first. There large lines have to be continued across multiple physical lines with the help of *continuation characters* (usually a `1` in the $6^{th}$ column or a `&` in the first). This is avoided in `C` but it is questionable whether this is really an advantage - depends on whether the programme contains mainly short or long lines.

[12] Some old compilers may not accept names that are this long!

letters, as also numbers. Since `C` is case sensitive, `aa`, `aA`, `Aa`, `AA` are all different variable names that can be used to denote different variables.

#### 1.3.4.2 Types of variables

When we do algebra, we do not care for types of variables. For us, a number is a number. Thus 5 and 5.3 have the same footing in our case. However, a computer handles integers (called `int`) and real numbers (called `float`) in very different ways and at very different speeds. This is why we need to use different types of variables to make our programmes optimal. Again, we may need to store names in *strings* and do different things with them. For that we need character variables (called `char`). The main problem is that inside the computer the variables are just linear arrays of binary bits, 1s or 0s. How does the computer know which variable is of which type? Again, different types of variables may need different numbers of bytes to store. So how does the computer allocate memory for storing these variables? This is why variables have to be *declared* at the beginning of the programme along with specification of types.[13] Each variable needs to be explicitly declared in `C` and all declarations must come before the first executable statement. [14]

Some of the common types of variables that we are going to use are:

[13] In `FORTRAN`, a variable name starting with `i`, `j`, `k`, `l`, `m` or `n` means integer and others imply real numbers and thus the explicit declaration of variables is optional.

[14] `C++` allows declaration of new variables anywhere inside the source code.

| ***Variable type*** | ***Keyword to declare the type*** | ***Example*** | ***comments*** |
|---|---|---|---|
| integer | `int` | `int bhoot;` | 4-byte in a 32-bit machine |
| real number | `float` | `float poot;` | for fractions using decimal points - 4-byte in a 32-bit machine. |
| big integer | `long` | `long int petni;` | 4-byte in a 32-bit machine[15] |
| big real number | `double` | `double mamdo;` | 8-byte in a 32-bit machine |
| very big real number | `long double` | `long double hutum;` | 16-byte in a 32-bit machine |
| single character | `char` | `char c;` | keeps the ASCII code coresponding to a *single* character |

Once declared, these variables can be used as often as needed inside the programme and their values can be changed, copied from one another and manipulated at will.[16]

#### 1.3.4.3 Handling variables in chunks - indexing / arrays

Often we need to handle large number of variables which belong in a group. Thus, for instance, a word is a collection of characters which come together. Again, manipulation of any kind of *indexed* data require arrays, e.g. matrices, $x - y$ tables with correspondence between the $i^{th}$ entries. Arrays of any of the above type of variable can be declared in the following manner:

[16] Why do we need the long variables? Because within limited space (number of bytes) the amount of information you can keep is limited. Bigger spaces allow bigger numbers as in the case of integers. In the case of real numbers, bigger numbers allow larger values as well as greater number of digits of accuracy.

```
float hablu[100];
int gobar[10][20];
char naam[20];
```

Here, `hablu` is an array of 100 real numbers[17]. They can be accessed from within the programme as `hablu[0]`, `hablu[1]`, $\cdots$ , `hablu[99]`. *Note that since we start from* 0*, the index goes up to* 99 *and not* 100*. In the declaration the* 100 *refers to the fact that there are* 100 *members in all.*

Similarly, `gobar` refers to an array of $10 \times 20 = 200$ integers which will be accessed from within the programme using the syntax `gobar[0][0]`, `gobar[0][1]`, $\cdots$ , `gobar[0][19]`, `gobar[1][0]`, $\cdots$ , $\cdots$ , `gobar[9][19]`. Thus a two-dimensional array will be implemented for us, although inside the computer, it is all the same.

The character array `naam` can be used to hold a string of characters, which may be anything, a word or sentence or some other code, limited to 20 characters in all.

#### 1.3.4.4 Initialisation of variables

If necessary, variables can be initialised as soon as they are declared. Thus:

```
int i=100;
```

will declare an integer variable and also fill it up with the value 100. Of course, this does not prevent us from changing the value of `i` from within the programme. But variables are manipulated using executable statements. Here, it is the compiler which directly fills up the memory location with the value 100, so it is not really an execution statement at all. [18]This is of limited usefulness.

However, this allows us to initialise character arrays in a very nice manner. Instead of giving the number of characters in the array explicitly, we could write:

```
char n[]="Naam Bolo";
```

[17]Not really. Inside the computer the name `hablu` refers to the address of the first member of the array. But for our present purpose, this is alright.

[18]Remember: `i=100;` and `i=100.;` are different. `100` is an integer and `100.` is a real number. Some compilers may allow the second form because `100.` has a way of being interpreted by an integer, but this is really mixed-mode operation. They should always be accompanied by a *type cast* - which we will discuss later.

Here, the array will have a size of 10 and be initialised like this:

| n[0] | n[1] | n[2] | n[3] | n[4] | n[5] | n[6] | n[7] | n[8] | n[9] |
|---|---|---|---|---|---|---|---|---|---|
| N | a | a | m | blank | B | o | l | o | NULL |

As you can see, *blank* is also a character (Hex: 0X20 in ascii code). Also, notice the NULL character at the end. It has an ascii code of 0X00 (i.e. all bits zero) and is used to mark the end of the string. In fact, by placing the NULL character anywhere, the string can be *terminated* to any size $\leq$ the size of the character array. Here, the size is not given, but the compiler, since it knows in advance what will go into the string, computes the correct size and creates the array.

A very similar thing could also be done with array variables of any other kind. We define a list of items with the help of second brackets. Thus:

```
float x[]={2.3,4.56,7.8,33.66};
```

will fill up `x[0]` with 2.3, `x[1]` with 4.56, etc.

### 1.3.5 Ordinary executable statements

#### 1.3.5.1 The meaning of the = sign

Unlike in mathematics, the = sign does not signify identity or equality. It is an assignment symbol and should better have been denoted by $\leftarrow$ . It means that the value of whatever is to the *right* of the = sign should be computed *first* and then the variable on the *left* should be loaded with that value. Thus there can be no computation on the left hand side

```
a = b+3;
```

is allowed, but not

```
a+5 = b+2;
```

This also allows some apparently peculiar constructs:

```
c = c+5;
```

This would be meaningless in an identity, but here the net effect is to increase the value of `c` by 5.

#### 1.3.5.2 Simple mathematical expressions

The *BODMAS* rule we learnt in school is followed in mathematical expressions. Some of the common operators are `+`, `-`, `*` (multiplication), `/` (division), `%` (modular division or the remainder left after the operand has been subtracted as may times as possible from the quantity on the left). There are also *right shift* and *left shift* operators, which we will not discuss now. Brackets can be nested to any level. We give an example:

```
halum = (a+20*(b+c*30)/9.332)/(a+b+34.5*c)+halum;
```

Lines like these form standard `C` code.

#### 1.3.5.3 Use of library functions inside mathematical expressions

If `math.h` is included in the header section and the `-lm` option is used during compilation, a huge array of pre-defined functions may be used inside the expressions. Thus:

```
halum = a*sin(hulum+0.033*cos(gablu));
```

will work as expected. *All angular references are in radians.* A list of some standard library functions is given below:

| | | |
|---|---|---|
| `sin(x)` | Sine of $x$ | math.h |
| `cos(x)` | Cosine of $x$ | math.h |
| `tan(x)` | Tangent of $x$ | math.h |
| `asin(x)` | arc sine or inverse sine of $x$ | math.h |
| `acos(x)` | arc cos of $x$ | math.h |
| `atan(x)` | arc tan of $x$ | math.h |
| `log(x)` | natural logarithm of $x$ | math.h |
| `log10(x)` | logarithm to the base 10 of $x$ | math.h |
| `exp(x)` | exponent of $x$ | math.h |
| `pow(x,y)` | $x^y$ where $x$ and $y$ are both double | math.h |
| `powf(x,y)` | $x^y$ where both are floats | math.h |
| `powl(x,y)` | $x^y$ where both are long doubles | math.h |
| `abs(i)` | absolute value of integer | stdlib.h |
| `fabs(i)` | absolute value of real number | stdlib.h |
| `floor(x)` | double having the value of integer nearest $x$ from below | math.h |
| `ceil(y)` | double having the value of integer nearest $y$ from above | math.h |
| `rint(h)` | double having the value of rounded integral value of $h$ | math.h |

Note: $\pi$ is denoted as `M_PI` and it is defined in `math.h`.

### 1.3.6 Preliminary input and output

The `printf` and `scanf` group of functions and many others provide the interface between the user and the running programme. We will give a minimal scheme for basic I/O.

#### 1.3.6.1 Reading in a string

After having defined a character array long enough to hold the maximum number of characters that our string will hold, values can be read in via the `fgets` function. Thus:

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

int main(void)
{
char halum[100]; /* This is the array of characters that will
hold the string */
```

```
fgets(halum,30,stdin);
return(1);
}
```

will read a string of characters from `stdin` (that is the keyboard where you type the input) upto a maximum 30 characters (if you type in more they will simply be ignored) and store them in the array `halum`. This process will occur only when you press the *Enter* key after typing. The ascii sequence denoting *Enter*, `0x0a` in UNIX and the `0x0d0a` pair in DOS, will be included in the string and at the end a *NULL* character (`\0`, actually the a byte with only zeros) will be appended automatically at the end. With this, the key depression sequence is now in the computer's memory [19]. This is only a string of ascii characters and will not denote any value even if the string is actually a meaningful string of numerals. The computer does not know how to differentiate it from any line of text!

#### 1.3.6.2 Reading in numerical values from a string

The `sscanf` function is used to read-in actual numerical values from a string. The syntax for reading an integer value will be:

```
sscanf(halum,''%d'',&bhoot);
```

Here `halum` is the string which holds the number in ascii form (read-in through `fgets`), `bhoot` is the integer variable which will hold the value [20] and the ''`%d`'' is a string which tells `sscanf` that it has to interpret the string in `halum` as an integer.

Let us try to understand `sscanf` better. It gets its input from a string. The name of the array which holds that string is the first argument (inside the brackets). The second argument is a string (which is why it is enclosed within '"') which gives the *format* i.e. how the input has been typed into the string kept in `halum`. Anything which begins with a `%` is a *placeholder* where you tell `sscanf` that it has to convert that portion of the string in a particular way. From the third argument onwards, you have to give a list

[19] The same `fgets` function can be used replacing `stdin` (standard input) by any file pointer to read lines from files, but that will come later.

[20] `&bhoot` represents the address of the beginning of the memory array which holds the variable `bhoot`. The `scanf` group of functions require the address, and not the name, to be given to the function. The reason for this is that in C, a function can only return one value; pointer variables (to come later) have to be used to return multiple values.

of addresses of variables where the converted values have to be stored. For instance, suppose you have typed `hulum molum khelum 432`. This value is now stored in the string `halum` after the execution of the `fgets` function. If you want the number 432 typed at the end of the string to go into an integer variable named `gablu`, the command will be:

```
sscanf(halum,"hulum molum khelum %d",&gablu);
```

Here, the conversion is done only when `sscanf` completes reading all the characters as given in the format string till it encounters the `%d`. Now it knows that it has to interpret the next few characters as an integer. It is clever enough to understand that the three characters `4`,`3` and `2` do form an integer. It does the conversion and stores it at the address that has been passed to it, i.e. inside `gablu`.

Let us try another example. We want to read in the values of two integers and we decide that the way we are going to type them in is → `The value of the first variable is 1234 and 678 is the value of the second variable`. Let the names of the two integers be `i` and `ii` respectively. The corresponding `sscanf` would be:

```
sscanf(halum,"The value of the first variable is %d and %d is
the value of the second variable",&i,&ii);
```

Note that here the first `%d` is matched with `i` and second with `ii`. There is no way the function really knows it has to do that. It simply matches the first `%d` with the first address which follows the format string and so on. It is the responsibility of the programmer to give the correct sequence of addresses. Note also that the first integer has four digits and second three. How does `sscanf` know that? It simply goes on including characters inside the conversion string until it faces something that will not go with an integer, here the blanks before the `and` and before the `is` are enough to tell `sscanf` where the numbers end[21]. Normally, no one likes to type in a lot of words along with their input values and the format string generally contains just a series of placeholders, separated by blanks if you want the inputs to be separated by blanks (or by commas if you want it that way).

[21]Something like `%3d` and `%4d` can be used to explicitly tell `sscanf` how many digits there are in the integers, but this is rarely used for reading-in numbers. Normal white space (blank, tab, etc) and other ascii characters which cannot be interpreted as parts of numerals are sufficient to delimit numbers.

#### 1.3.6.3 The placeholders

Different kinds of variables require `sscanf` to interpret parts of the input string in different ways. This necessitates the use of things like `%d`. We describe some common formats:

`%d` → integer. The number of digits can be explicitly declared with syntax like `%5d`.

`%f` → floating point number. Decimal points will be interpreted correctly, but if required, the structure of the number can be given explicitly. Thus `%14.7f` means that the total length of the number (including sign if negative or `+` is given explicitly) is 14 characters and the decimal point is followed by seven numerals.

`%e` → Floating point number in scientific notation with an `e` representing the exponent, e.g. `14.78e5` which means $14.78 \times 10^5$ .

`%g` → Floating point number in either plain or scientific notation depending on which ever it encounters. [22]

`%c` → a single character. Here only one character will be read in from the string.

`%s` → a string of characters[23].

`%[^ ]` → This is used to tell `sscanf` to read in a substring from another string. The substring should be considered to have ended if we encounter a blank. The `^` sign signifies that anything that is *not* blank. For example, if we stored into halum the string : `There is a full moon tonight and the Fairy Queen-Mother is paying us her 3rd visit`, Then we could read in the strings `moon`, `Fairy Queen` and the integer 3 through the following programme:

```
#include <stdio.h>
#include <string.h>
int main()
{
```

[22] This is actually more relevant when the format is used for output with `printf`, which we will discuss later.

[23] Note that for the `%c` and `%s` formats, we already have the characters in the input string when we use `sscanf`. So, if we like, we can just use the proper index to get the value from the array.

```
char halum[80];
char first[20], second[20];
int num;

fgets(halum,80,stdin);
sscanf(halum,"There is a full %[^ ] tonight and the
%[^-]-Mother is paying us her %drd visit",first,second,&num);

printf("first is -> %s\n", first);
printf("second is -> %s\n", second);
printf("num is %d\n",num);
return(1);
}
```

Here the `printf` function has been used to print out the values. We will discuss `printf` below. Notice, however, that when the strings are read-in, the addresses of `first` and `second` are apparently not given. Actually, `first` and `second` being arrays, the name `first` without any third brackets following is a shorthand for `&first[0]`, which is the address of the first character in the array. So there is no mistake.[24]

#### 1.3.6.4 Output with `printf`

The `printf` function is the workhorse of `C`, since whatever we do, we need to get some output! The syntax of `printf` is very much like `sscanf`, but there is no string here and the actual variables and not their addresses have to be passed to it. The format string works just like in the case of the `scanf` group of functions. Thus

```
printf(“This is the value of bhoot -> %g and poot is the
character %c\n”,bhoot,poot);
```

will print out the format string with the values of bhoot (a float) and poot (a character) in the correct places[25]. The `\n` represents the end-of-line or *newline* character. If we want the cursor to go to the next line after printing the

[24]It is generally not a good idea to start a read-in sequence with `fgets` because the person running the programme will have to know what exactly is written in the code to enter data correctly. This is why it is always good practice to put in a `printf` line before the `fgets` which will tell the user what and how exactly to type in the input. This `printf` will, however, have no bearing on how the next lines of code work.

[25]Note we are using `bhoot` instead of `&bhoot`. The addresses need not be passed explicitly to the `printf` group of functions. The reason for this is that functions (subroutines)

output, the newline character will have to be included in the format string[26].

#### 1.3.6.5 A simple programme showing input, output and some elementary processing

We give a programme below which will read in two floating point numbers and print out some derived values:

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

int main(void)
{
char hondal[80]; /* holds the input string */
float a,b,c;

printf(“Give me the values of a and b separated by blanks:  “);
fgets(hondal,80,stdin);
sscanf(hondal,’’%g %g’’,&a,&b); /* Values are loaded into
variables a and b */
c=a+b;
printf(“The sum of %g and %g is %g\n’’,a,b,c);
printf(“The sine of c is %g\n’’, sin(c));
/* The line below illustrates how evaluation can be done inside
the printf function also */
printf(“sin(%g)+cos(%g) is %g\n’’,a,b,sin(a)+cos(b));
printf(“Thank you!  Good bye!\n’’);
return(0);
}
```

Now we can compile this programme (remembering the `-lm` option!) and run it. We are now equipped to handle basic input and output of different data types.

---

in `C` can return only a single value directly to the calling programme. If multiple values have to be somehow shown to the calling programme, the only way is to use *pointers* which allow direct insertion into the calling programmes memory. But pointers will come later!

[26] Try running a programme with and without the `\n`!

### 1.3.7 Decision making in C

The great power of digital computers comes from their ability to perform logical operations. This enables correctly written programmes to have remarkable decision-making capability. To use this functionality, we need to understand some logical operations.

#### 1.3.7.1 Logical operations can have only two outputs

The kind of logic we discuss here does not allow any *gray* areas. A statement is either `TRUE` or `FALSE` which represent values of `1` and `0` respectively. All logical operations return either of these two values. Commonly logical operations are used to compare two normal variables. Thus:

`a == b` will return a `TRUE` only if `a` and `b` are equal and `FALSE` otherwise (Note the use of the double `=` sign).

`a < b` will return a `TRUE` only if `a` is less than `b` and `FALSE` otherwise.

`a <= b` will return a `TRUE` only if `a` is less than or equal to `b`.

`a > b`, `a >= b` similar to the above.

`a != b` will return a `TRUE` only if `a` and `b` are different.

Logical expressions can be combined using the *AND* (`&&`) and *OR* (`||`) operators. Thus:

`(a == b)&&(c != d)` will return `TRUE` only if `a` is equal to `b` and `c` is different from `d` $\cdots$ both conditions have to be satisfied together.

`(a < 10)&&(a > 5)` will return `TRUE` only if `a` is greater than `5` and less that `10`. Here, the limits are not included in the range. If you want to include the endpoints within the range, you have to use the `>=` or `<=` operators.

`(a > 10)||(a < 5)` will return `TRUE` for all values of `a` outside the range `5-10`, endpoints not included.

Any number of such conditions can be combined to produce very complex logical statements, which will, however, produce either a `TRUE` or a `FALSE` at the output.

#### 1.3.7.2 `if` and `else`

The most important decision-making construct is `if`. The structure is:

```
if(··· some logical expression ··· )
{
··· some executable statements ···
}
else
{
··· some executable statements ···
}
```

This is how it works ··· if the logical expression returns `TRUE` then the statements within the braces are executed and the statements following the `else` clause are jumped over and execution resumes from below the closing brace. If, however, `FALSE` is returned, then the statements following the `if` clause are skipped and the statements following the `else` clause are executed and execution resumes from below. The programme below will illustrate the above:

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

int main(void)
{
char input[1];
printf("Give me y or n:  ");
fgets(input,10,stdin);
if(input[0] == 'y')
{
printf("Hei kotta, haan bolecho go!\n");
}
else
{
printf("Hai Hai!  naa kore dile!\n");
}
return(1);
}
```

Try out this code and see what it does. The `else` clause may be left out if not required. Thus:

```
if(a==b)
{
c = 2*a;
}
```

will simply ignore the statements following the `if` clause if `a` is not equal to `b`. `if`s may be nested (put one inside the other) to any level.

One interesting variation of the syntax occurs when there are multiple `if-else` clauses nested within each other:

```
if(a == 5)
{
b = 420;
}
else
{
if(a == 20)
{
b=840;
c=22.5;
}
else
{
if(d == 9)
{
halum = 2;
}
else
{
printf(“aar parinaa bapu!\n”);
}
}
}
```

This can be expressed in a much more readable form as:

```
if(a == 5)
{
b = 420;
}
```

```
else if(a == 20)
{
b=840;
c=22.5;
}
else if(d == 9)
{
halum = 2;
}
else
{
printf(“aar parinaa bapu!\n”);
}
```

One of the potentially largest uses of the `if` clause is for the construction of loops. Thus, if we want to repeat some procedure 10 times, we could proceed like this:

```
#include <stdio.h>
int main(void)
{
int bhombol;
bhombol = 9;
goop:  if(bhombol >= 0)
{
··· some executable statements ···
bhombol--; /* reduce bhombol by one */[27]
goto goop;[28]}
```

[27] `++` and `--` are two operators which can be used to increment or decrement a variable by `1`. Thus, for us, `a = a+1` and `a++` (or `++a`) are the same, but since many processors have special OP-codes for high-speed increment and decrement, `++` and `--` may be implemented differently, and possibly at higher speeds than `a = a+1`. A somewhat similar syntax is `a += 5` for `a = a+5`.

`a++` and `++a` are different. We will generally not need to keep this difference in mind because we are not going to use constructs where the difference will become noticeable. However, if a variable `a` is given a value of five, then `c = a++` will leave `c` with a value of five and `a` with a value of six, but `c = ++a` will leave both with a value of six because in the second case `a` will have been incremented before being assigned to `c`.

[28] The `goto` statement is equivalent to the `JMP` OP-CODE and causes an unconditional jump to the statement containing the label (here `goop`). One should try to use this statement as little as possible because this causes the programme to loose its ‘structure’ and future debugging and understanding becomes difficult. Somewhere I read a definition of goto like this: *`goto` → a statement that enables structured programmers to complain about unstructured programmers!*

```
}
return(1);
}
```

However, there are more suitable constructs to handle loops of different kinds, which we discuss below.

#### 1.3.7.3 The `while` loop

Consider the following piece of code:

```
i = 5;
while(i > 0) /* remember, no ; at the end here */
{
...;
...;
--i;
}
```

Here, the portion of code between the braces will go on being executed in a loop and at the beginning of each run, the condition will be tested. After the loop exits, the code following the closing brace will be executed. Thus in this case the loop will run five times. Note here that the initialisation of `i` has to be done outside the loop and it has to be decreased (or manipulated in some other way) inside the loop. Otherwise we will be left with an infinite loop because `i` will not change at all. This makes the `while` loop the most general loop in C, but for the kind of work shown in the above code, where the exact initialisation and manipulation of the variable is known in advance, we will see that the `for` loop is better. The `while` loop really comes into its own when the loop has to test for an externally supplied condition, e.g.

```
fgets(input,79,stdin);
while(input[0] != 'q')
{
...;
...;
fgets(input,79,stdin);
}
```

Here the programme does not know in advance how many times it has to loop, but continues looping as long as you do not type in a `q`.

#### 1.3.7.4 The `do while` loop

This loop is very similar to the `while` loop, with the condition placed at the end of the executable statements. It is checked at the end also, so, even if the looping condition is not satisfied, the loop *always runs at least once*. This is the main difference between the two forms of the `while` loop. Which form to use depends on the need and preferences of the programmer.

```
do {
···;
···;
fgets(input,79,stdin);
while(input[0] != 'q'); /* There must be a ; here!  */
```

The do-while construct is not used very often.

#### 1.3.7.5 The `for` loop

In the case of loops which depend on some variable which needs to be initialised and manipulated in a way which the progamme knows in advance, the best loop to use is the `for` loop. This loop takes three statements within its brackets; the first one for initialisation, the second one for the condition, the third one for changing the variable used in the test condition. Thus, the first `while` example that we saw can be expressed much more concisely as:

```
for(i=5;i<0;--i) /* no ;!  */
{
···;
···;
}
```

Here, the *action* part of the loop and the initialisation, checking and control variable manipulation parts are neatly separated and the loop is easy to understand and compact. This is the most commonly used loop and is used universally whenever vectors, matrices and known iterations are handled. Of course, any loop can perform the functions of any other if used correctly ··· all this is just for convenience[29].

[29]The `for` loop allows initialisation of more than one variable in the first statement. In that case commas have to be used to separate these statements. Similarly, the third place can also contain more than one statement. Thus something like `for(i=5,j=6;(i>2)&&(j<10);--i,++j)` is allowed.

### 1.3.8 User defined data types

Users can define their own data types in C! These data types may be the same as those already defined but with different names, or much more complex entities.

#### 1.3.8.1 `typedef`

We may want to give a new name to an already known data type. Thus

```
typedef int hallu;
```

will create a data type called `hallu` which is nothing but an `int`. This has to be placed either before the start of `main` or among the variable declarations, before the first use of `hallu`. Thus `hallu bhallu;` will declare an integer named `bhallu`. Why do we use `typedef`? One of the reasons is when we are not sure whether a particular group of variables will change type as we keep modifying and debugging a programme. Say for instance we know a group of variables `a, b, c, d` will be of the same type, which seems to be, at the outset, to be `int`. Instead of declaring all these variables as `int`, however, we declare them to be of type `hallu`:

```
hallu a,b,c,d;
```

Some time later, if we decide that these variables should better be of type `float`, all we have to do is to change the `typedef` declaration to `typedef float hallu`. A lot of retyping is saved and chances of inconsistent data type in a large programme is reduced. It is very good practice to use `typedef` liberally. Another very useful application of `typedef` is in conjunction with `structures`, which we discuss below.

#### 1.3.8.2 Structures

One of the most beautiful and useful abilities of C is to allow the progammer to define complex data types, which allows one to *encapsulate* a large number of variables, often of different types, into a composite whole. These composites, of any complexity[30] are called structures and declared with the keyword `struct`. Suppose we want to store the age of a person in a `struct` called `age`. The declaration and definition of the `struct`

[30] `structures` may hold other `structures`.

```
typedef struct {
int years;
int months;
int days;
} age;
```

which will often be placed at the top of the programme, before the `main`, will define a data type `age`, which is a complex construct with three *components*, the integers `years`, `months` and `days`[31] .

It is very important to understand what we have done. `age` is not a variable, it is a new data type and does not require any memory by itself. When we need to actually store the age of an individual, however, we declare an actual variable:

```
age gablur_boyosh;
```

Now `gablur_boyosh` becomes a new variable of type `age`. Memory required to hold the three integers is now allotted and from now on we can deal with `gablur_boyosh` as a valid variable. How do we access the inner variables? How do we get at the components of a vector $\vec{A}$? We use dot products like $\vec{A} \cdot \hat{i}$, etc. A similar syntax is used here also - `gablur_boyosh.years` will allow us access to the first integer inside `gablur_boyosh`. Let us see an example programme:

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

typedef struct {
int years;
int months;
int days;
} age;
```

[31] Actually the `typedef` need not be used. We could have just written:

```
struct age {
int years;
int months;
int days;
};
```

However, if declared in this way, the word `struct` will have to be added in front of every instance of declaration of variables of type `age` ... `struct age hallu;` ... This can be avoided by the use of the `typedef` declaration, and `age` is treated by the compiler at par with any other data type like `int`.

```
typedef struct {
char name[80];
float height;
age umar;
} student;

int main(void)
{
char input[20];
float lombu;
student chhana;

printf(“Give me the name of the chhatra:  “);
fgets(chhana.name,79,stdin);
printf(“Give me the height of %s:  “,chhana.name);
fgets(input,19,stdin); /* note how we leave one character free
in the strings to accommodate the null character at the end */
sscanf(input,”%g”,&lombu);
chhana.height = lombu; /* could have read in chhana.height
directly also */
printf(“Give me the age of %s as yyyy:mm:dd -> “,chhana.name);
fgets(input,19,stdin);
sscanf(input,”%d:%d:%d”,&chhana.umar.years,&chhana.umar.months,
&chhana.umar.days);

chhana.name[strlen(chhana.name)-1] = ’\0’;

printf(“The name of the chhatra is %s and height is %f and age
is %d years, %d months and %d days\n”, chhana.name,
chhana.height, chhana.umar.years,
chhana,umar.months, chhana.umar.days);

return(0);
}
```

The programme is basically starightforward. We could have declared an array of students by `student chhana[100];` and put in the values one after the other through a `for` loop. Note a small point however. The fgets function includes the `\n` character (end of line) as part of the string (because you did type in an *enter* to complete the input!). To eliminate this character, we

have to put in the end-of-string null character (`\0`) in its place. The `strlen` library function returns the number of characters in the string, including the `\0`. Convince yourself that what we have done actually serves our purpose (remember, array indices start at 0).

Structures help us to organise data into compact packets. Use them as much as possible[32].

### 1.3.9 Functions

Functions are subroutines.

#### 1.3.9.1 What they do

When a sequence of operations needs to be repeated in many different places, it is convenient to organise them as a separate block and jump (like `goto`) to its beginning when needed. But an ordinary `goto` will need a reverse `goto` at the end of the block to come back to the main sequence. This is where we face a big problem. We know where the start of the block is, so we can jump there. Where do we need to come back? Just below the point where we had departed from the main sequence. This address will vary as we call the same block from different placed in the main programme. So what address should we give to the `goto` at the end of the block? How do we keep changing it without recompiling the code in some weird fashion? To solve this problem, subroutines have been added to computers at the very machine language level. Instead of a simple jump, we use a CALL (in assembly language). On encountering a CALL, the CPU first computes the address where it will have to come back and places that address in a region of memory called the *stack*[33]. Then it jumps to the start of the called block. At the end of the block, instead of a regular jump, we have a RET - an instruction which tells the CPU to find the address to return to from the stack and then jump to

[32] *Intelligent* structures called *classes* which include variables as well as programme code (in the form of subroutines) are of fundamental importance in C++ programmes.

[33] A stack is a region in memory, usually near the end of RAM, which is accessed in the form of a LIFO via a register called a stack-pointer. LIFO (last in first out) is the kind of arrangement we have when we stack logs of wood; the last log to get onto the stack is the first to be *popped* off. In the case of computers, the stack-pointer is loaded with the address of the beginning (the last memory - called high memory as the address is higher) of the stack. When values are *pushed* onto stack, the bytes to be saved are placed one after the other in the addresses *pointed to* by the stack -pointer, which is decremented after each save. Thus if the first byte is placed at address FFFFH, the next will be at FFFEH and so on. During *popping*, bytes are loaded in and the stack-pointer is incremented. This is how the LIFO is implemented. FIFOs are also used - in the form of something called a *pipe* - these are used to join input of one programme with the output of another programme.

that address. When organised in this way, such a block is called a *subroutine.* In C, subroutines are called *functions.*

#### 1.3.9.2 Function calling arguments

Functions may or may not take *arguments* which are values passed to it by the main programme. For instance, the code snippet `sin(x)` inside a main programme (or, may be, another function) calls the function `sin` (which is already written by experts and supplied in a half-digested compiled form (object files arranged in an library archive) in the math library with the value of the variable `x`. How does the function actually get hold of the value of `x`? The main programme actually places the values of the arguments (may be as many as you like) in the stack and calls the subroutine, which reads in the values from the stack. Thus the subroutine needs to know how many values and of what kind it has to read-in from the stack. More on this later.

#### 1.3.9.3 Return values

`sin(x)` returns the value of the sine of `x` to the main programme using a process similar to the transfer of argument values. Do all functions need to return values? Not at all. A function may be programmed to do anything - say for instance when the function called `gnatta` is called from the main programme, it activates a robot to do what its name implies on my head. It is quite effective without taking any argument or returning any computed value. If it does not take any argument, it will be called with an empty argument list - `gnatta()` - so that the compiler knows that it is actually a subroutine call. *A very important thing to remember about functions in C is that only one variable can be returned, there is* ***no direct way*** *to return multiple results with a single call.*

#### 1.3.9.4 Function declarations

The main programme needs to know[34] which functions it may call and the types and numbers of the arguments and the return type. This is why a *prototype* is declared at the beginning of the function, along with the variable declarations. Thus:

```
float gablu(float,float,int); /* note the ; at the end */
```

[34]Strictly speaking, if the function is *defined* before the main programme, this declaration is not necessary, because the compiler already knows about the function before it meets the first instance of its invocation, but it is good form to always declare functions.

will declare that the main programme (or whoever will be using `gablu`) will need to handle a function called `gablu` which will take two `float`s and one `int` (in that order) and return a `float`[35].

#### 1.3.9.5 Function definitions

Where is the code for the function? It is placed outside the `main`, within its own pair of braces. The first line will be very like the line in the declaration, with actual names for the arguments and *no semicolon at the end.* We illustrate the idea with a small `main` and and a function `ulta_bujhli_ram` which returns the negative of the calling argument:

```
int main(void)
{
float lombor;
float borlom;

float ulta_bujhli_ram(float); /* this is the declaration */

printf(“Give me a number:  “);
scanf(“%g”,&lombor); /* safer always to use the fgets-sscanf
route!  */
borlom = ulta_bujhli_ram(lombor);
printf(“The ulta of %g is %g\n”,lombor,borlom);
return(0);
}

float ulta_bujhli_ram(float x) /* this is the definition - the
real thing!  */
{
float y;
y = -x;
return(y);
}
```

[35]Note that the argument list within the declaration does not need to have any name (it is not wrong to do that - in fact, it is said that it allows some compilers to do better type checking). This is because the subroutine actually does not know the values by the names used by the calling programme - it just pops values off the stack according to the number and types of the variables - the names that the function uses to handle these variables are entirely its own - in technical parlance within the *scope* of the function only.

There are some points to be noticed here. The variables declared within the argument list of the function definition need not be declared within its body again[36]. The other thing to note is the syntax used for returning the return value. Unlike FORTRAN, the name of the variable being returned (or an expression) has to be explicitly given. `return` is not a function, it is part of the language - so the bracket is not necessary :- `return -a;` will do.

#### 1.3.9.6 Returning multiple values from functions via structures

What if I need to return more than one value? For instance we may write a function onko which returns the sum, difference and product of two numbers. Here we need to read-in two numbers and return three. But we can return only one variable. Our structure comes to the rescue here. A structure object may hold as many variables as it likes within itself, but still is only one (complex) variable! See the code below:

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

typedef struct
{
float jog,biyog,gun;
} uttor;

int main(void)
{
char input[80];
float a,b;
uttor halum;
```

[36]This is not strictly true. It is possible to declare the function without any argument list - just the brackets. Then within the definition something like this could be done:

```
float ulta_bujhli_ram(a)
float a; /* before the opening brace!  Declare all the arguments one after
the other.  */
{
return(-a);
}
```

This is generally not a good idea because the compiler does not know from the declaration whether the calling parameters are of the correct number and type, but if the list of arguments is very long, this may help to make the code more concise.

```
uttor onko(float,float);

printf("Give me two numbers:  ");
fgets(input,79,stdin);
sscanf(input,"%g %g",&a,&b);
halum = onko(a,b);
printf("The answers are %g, %g and %g\n",
halum.jog,halum.biyog,halum.gun);
return 0;
}
uttor onko(float a, float b) /* these a and b are different
from those of the main!  Think of scope difference.  */
{
uttor temp;

temp.jog = a+b;
temp.biyog = a-b;
temp.gun = a*b;

return(temp); // bracket or no bracket does not matter.  */
}
```

This method will always work. However, the values are being copied to and fro here and if the number of variables is very large (huge array, say), this is not the most efficient method. The standard method of returning multiple values is through *pointers*.

Can a function have a return value of type `void`, i.e. not return anything at all? Our `gnatta` example is one such. It would probably be declared `void gnatta(void);`. Such functions can also return values through *pointers*.

### 1.3.10 Pointers

*Pointers* form one of the most risky and powerful constructs in C. They can be used to write virus programmes! By the same token many bugs kreep into programs through wrong use of pointers. They are very useful but should be used with care.

#### 1.3.10.1 Location, address and content

The actual physical circuitry in a RAM or ROM where a byte of data resides (in the form of high and low voltages representing bits) is called the *location*.

The sequence of voltages at the location forms the *content* of the location. To access the contents of the location, a pattern of bits (voltages) has to be applied to *address lines* of the memory module. This pattern is the *address* of the location. Thus, given an address, the contents of a location of memory may be read or altered. In fact, when we declare a variable by a particular name, say `a`, we ask the compiler to set apart a set of contiguous bytes (depending on how many bytes are needed to keep a particular data type) and to refer to its starting address whenever we perform some operation on or with `a`.

#### 1.3.10.2 `&x` is not a variable

If `x` is a variable, `&x` refers to its address. Who determines its address? The compiler and loader[37]. Thus something like `&x = 2000;` will not work because the address is already fixed. Thus `&x` will always appear on the r.h.s of an expression, when we need to learn the address that the compiler has allotted to it[38].

#### 1.3.10.3 Pointers are variables

A pointer is a variable and so its contents may be changed from within the programme. It holds the address of another variable. Pointers are declared by putting a `*` in front of the variable name, after which the address of another variable may be loaded into it:

```
float a; /* just an ordinary variable */
float *p; /* p is a pointer variable - right now it does not
hold anything */
p = &a;
```

Here we have declared a pointer variable called `p` and loaded it with the address of the variable `a`. Now let us forget about `a`. We have only `p`. Can we retrieve the value of `a`, i.e. the contents of the address that is in `p`? This

[37] The *loader* puts in the actual values of the memory used and loads the programme in the correct place in RAM when the programme is executed. The compiler uses symbolic representations to denote momery addresses. This is important in a multi-user, multi-tasking OS where the portion of memory that will be available is not known beforehand. It depends on how many users are logged in, how many jobs are running, how much memory is used by the jobs, etc.

[38] It may not be the compiler only which allocates memory. *Dynamical allocation* may be achieved using `malloc`. In that case also, `&x` will refer to the memory allocated.

content can be accessed through the symbol `*p`. Thus `printf("%g\n",*p);` will give us the value of `a`.

#### 1.3.10.4 Meaning of pointer type - pointer arithmetic

Since a pointer holds the address of a byte, why do we need the `float` in `float *p;`? The fact is that there is nothing in the address that is contained within the pointer that can help the program to understand the type of the variable whose address will be contained inside. Why do we need to know the type in the context of pointers, when we already must know the type $\cdots$ as the programmer? The answer is that there is a very powerful set of operations called *pointer arithmetic* which we can do with pointers. If we have an array, and if a pointer `p` is assigned the address of the first byte of the array (something like `p=&a[0];` or `p=a;`, which is the same thing), `p+1` will refer to the next member of the array. Similarly `++p;` can be used to increment `p` so that it points to the next member. But different types of variables have different sizes[39]. Then how does `p` get incremented properly so that it really skips the requisite number of bytes so as to point to the next member? This is where the information regarding the type of the variable whose address is being kept in the pointer (the *type* of the pointer) is used[40].

#### 1.3.10.5 Arrays and pointers

Arrays are always accessed through pointers. If we declare `int bb[10];`, then space for keeping ten integers is allocated and the address of the first byte is returned as a pointer called `bb` (without the third brackets). Now `bb[i]` is just a shorthand notation for `*(bb+i)`. Now if we declare a new pointer like this: `int *pp;` and set `pp` equal to `bb`, then we can use `pp[i]` just like `bb[i]`. However, if `pp` is not set equal to `bb`, it does not contain the address of any valid variable and something like `pp[10]` will point who knows where and may lead to a segmentation violation error. For multi-dimensional arrays defined with something like `float halum[20][10];`, we actually have 20 one-dimensional arrays, each of which has a valid starting address, and the array of such addresses forms another array, which has its starting address kept in halum! Incrementing the first index (the one on the left) causes a jump over $10 \times sizeof(float)$ bytes. Incrementing the second index causes a shift to the next variable.

[39] The `sizeof` function can be used to find the number of bytes required for a particular data type. Thus `sizeof(int)` will return the number of bytes required for an `int`.

[40] Notice the consistency of the notation. Since `*p` implies the value kept at the address kept in `p`, something like `char *p;` makes sense $\cdots$ `p` is such a pointer that it keeps the address of something the contents of which is of type `char`!

#### 1.3.10.6 Returning multiple values from functions with pointers

The preferred way of returning multiple values from functions is through pointers. What the calling programme does is to pass, along with the normal arguments, some pointer arguments which contain the addresses of some variables (of the calling programme) in which the function can write the results directly. `return` is not needed at all[41]! We illustrate this with the help of a function `onko`, which takes two floats and calculates the sum, difference, product and quotients and writes them directly into the memory of the calling progamme. Beware that the function is being allowed to write directly into the memory of the calling function. This may be the cause of disaster with badly written code.

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

int main(void)
{
float a,b;
float sum,diff,prod,quot;
char input[80];

int onko(float,float,float*,float*,float*,float*);

printf(“Give me a and b:  “);
fgets(input,79,stdin);
sscanf(input,”%g %g”,&a,&b);

if(onko(a,b,&sum,&diff,&prod,&qout) != -1)
{
printf(“sum = %g, difference = %g, product = %g and quotient =
%g\n”,
sum,diff,prod,quot); /* the results are already inside these
variables!*/
}
```

[41] `return` is not needed in these cases, but it is always good practice to have a `return` statement with the help of which it may be possible to signal (through the value of the returned variable, which is generally set to an `int` in these cases) special conditions, e.g. whether there was any error or problem during some step (files failing to open, zero in the denominator which has been properly detected), etc. The calling programme may then be programmed to look at the return value and take corrective steps, if necessary.

```
else
{
printf(“sum = %g, difference = %g and product = %g, the
quotient blows up or is undefined!\n”, sum,diff,prod);
}
return(0);
}

int onko(float x,float y,float *p,float *pp,float *ppp,float
*pppp)
{
*p = x+y;
*pp = x-y;
*ppp = x*y;
if(y != 0)
{
*pppp = x/y;
return(0); /* everything fine!  */
}
return(-1);
}
```

We take another example, a function that interchanges two variables inside the calling programme without returning anything.

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

int main(void)
{
int a,b;
char input[80];
void myswap(int*,int*);

printf(“Give me two integers:  “);
fgets(input,79,stdin);
sscanf(input,’”%d %d”,&a,&b);p
printf(“a is %d and b is %d before calling myswap\n”,a,b);
myswap(&a,&b);
printf(“a is %d and b is %d after calling myxwap\n”,a,b);
return(0);
```

```
}

void myswap(int *x,int *y)
{
int tmp;
tmp = *x; /* keep it safe */
*x = *y;
*y = tmp; /* ere mundu ore ghare!  */
}
```

Often functions are called with array names as parameters. Since array names are actually addresses, any change that the function makes to the array as it sees it will be reflected in the array in the calling programme. This is different from calling with non-array variables, in which case the variables are local to the function, even if the name is the same. We illustrate this difference below:

```
#include <stdio.h>
void main(void)
{
int a, b[] = {1,2};
void hamlu(int,int*); /* going to pass array through the second
argument */
a = 5;
printf("a = %d, b[0]=%d,b[1]=%d\n",a,b[0],b[1]);
hamlu(a,b);
printf(" after calling hamlu a = %d,
b[0]=%d,b[1]=%d\n",a,b[0],b[1]);
}

void hamlu(int a, int *bhombol)
{
a = 20; /* trying to change a of the main programme should not
work */
bhombol[0] = -420; /* should change b[0] of the main!  */
}
```

### 1.3.11 File access

Data stored in a hard disk, CDROM or any other secondary medium is generally organised as a *file*. We can read from and write to files just like we

do from the keyboard and screen. Any programme which deals with a large amount of data necessarily uses files. One simple way to store data in a file is to use *redirection*:

```
$./gablu > gablu.out
```

This will write whatever output was being made to the *standard output* (screen mostly) into `gablu.out`. Similarly

```
$./gablu < gablu.in
```

will make `gablu` read-in from `gablu.in` on each invocation of `scanf` (which reads in from `stdin` - the keyboard mostly)[42]. However, this is stop-gap method which will not work if many files need to be accessed. Proper file access is done through dedicated functions.

#### 1.3.11.1 Why we need a `FILE` structure

In any advanced Operating System, data is *buffered* in RAM. Access to secondary storage devices is much slower than RAM access. In the case of write instructions, the hard disk drives are organised in such a way that whole chunks of data can be writted somewhat fast (using something called DMA, or direct memory access), but individual writes are very slow. It is profitable to store the data in some place in RAM and once sufficient data has been stored, a single block transfer operation can be executed to load the data into disk. This two-stage transfer is managed by the OS, the programme does not know about this. Thus when we write our data, we need not keep these matters in our mind. Our programme writes data as and when necessary. This means that when we write to a file, some of the data is in the hard disk and some in RAM, waiting to be transferred in its own time[43]. But this introduces complications - much accounting information and data has to be kept so that the data is not sent anywhere at random.

Again, when we read-in data from a file, a clever trick by the OS often (but not always) causes a big improvement. It is often seen that most big

[42]There is also something called *standard error* - `stderr` which can be accessed via `fprintf`. This also prints to screen. It can be redirected by using `2>`. An example of a command which makes `gablu` read-in from `gablu.in`, redirect normal output to `gablu.out` and redirect error output to `gablu.err` is:
`$./gablu <gablu.in >gablu.out 2> gablu.err`

[43]This is one of the reasons why there is file corruption and data loss when you switch off the computer without going through a proper shut-down procedure. The part on the disk is secure, but the part in RAM is lost!

programmes will read in data from a file in sequence. So if OS can read-in a big chunk of data in a *read-ahead-buffer* then it is just likely that during subsequent reads, the OS can just supply the data from the RAM. So more space is needed. Again, somewhere we must have a record of just where in the file we are. When we demand the next piece of data, it must come from the proper place in the file. So the current record number, or something like that must be kept.

All these things are managed through a special kind of structure defined in the standard headers as a `FILE`. The common operations are performed through pointer variables which point to `FILE` structures (the capitals are necessary - C and UNIX is case-sensitive) . They have to be declared at the top of the programme (one for each file):

```
FILE *in, *out, *halum;
```

#### 1.3.11.2 `fopen` and `fclose`

To *open* a file, a link must be made between an actual file (with a name) on the disk and an already declared `FILE` pointer. The name of the file and the kind of access (read or write for us now, but many more options exist) have to be supplied:

```
in = fopen(“hallu.gallu”,”rt”); /* a text (ascii) file opened
for reading */
out = fopen(“gablu.bhombol”,”wt”); /* a text file opened for
writing */
```

After working with the files, one should close them:

```
fclose(in); fclose(out);
```

There is a very large set of options which may be used with `fopen`. The type of the file may be *binary*, one may *append* data at the end of the file, etc. Right now, we need only the options discussed. Remember one thing however. If a file is opened in *write* mode, the previous contents will be erased. In the *append* mode (“`at`”), previous data will remain.

#### 1.3.11.3 Reading and writing into files

`stdin` is nothing but a file pointer, already declared by the OS and linked to the *file*[44] (already opened), which is nothing but the keyboard. To read data from an already opened file, one has to only replace `stdin` by the corresponding file pointer in `fgets`:

```
fgets(input,79,in);
```

As simple as that! There is another more standard way. Just follow the syntax for `sscanf`, but use `fscanf` and instead of the name of the string, use the file pointer:

```
fscanf(in,''%g %g'',&a,&b);
```

For writing to files, the best option (there are a very large number of ways of handling files!) is to use `fprintf`:

```
fprintf(out,''The result is %d\n'',halum);
```

There is one problem with buffering. If we have a programme which keeps writing some data to a file which we plot in real time with the help of another programme, we cannot always wait for the buffer to be flushed. A `fflush(out);` after a write will always ensure that the data is actually immediately written to disk. This is very important for programs running experiments.

#### 1.3.11.4 Reading in data from a file without knowing the number of lines beforehand

When we have a programme which does something with data read-in from a file (a regression fit, say), we want to write it in a manner that does not necessitate our knowing the number of lines beforehand. It is preferable that the programme should be able to determine the number of lines in the file by reading the file itself. For this purpose, we use the return value of the `fgets` function. The function returns a pointer to the character array where the data is getting stored when there is data to store. When it encounters an end-of-file, it returns a `NULL` pointer. This can be used to implement a flex-

[44] In UNIX, almost all i/o devices are treated as files. The actual hardware is attached with the help of special kernel modules to *device-special-files* in the `/dev` directory.

ible input mechanism as shown in the following example, where we assume that each line in the file has two data (`x` and `y`) which are separated by tabs:

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

int main(void)
{
char input[80];
float x[100],y[100]; /* we will not deal with more than 100
pairs of data here!*/
int n;

FILE *bhalluk;

bhalluk = fopen(“bhutum.in”,”rt”);
n = 0;

while(fgets(input,79,bhalluk) != NULL)
{
sscanf(input,”%g\t%g”,&x[n],&y[n]); /* input already contains
the line!  */
++n;
}

printf(“The number of data is %d\n”,n);
··· ; /* use the data as you like here!  */

return(0);
}
```

#### 1.3.11.5 Random numbers from /dev/urandom

Among the various devices that the kernel handles (and provides) and allows us to access just like files, are two very interesting ones, `/dev/random` and `/dev/urandom`. They provide random bytes[45]. Since we use bytes to make

[45]Just type `$cat /dev/random` and `$cat /dev/urandom` and see how random bytes look when interpreted as characters. These random bytes are generated using an *entropy pool* which is left behind in the memory by different device drivers once their work is done. `/dev/random` produces the best quality random bytes, but needs real events so that device drivers are activated, and so may stop working and wait for some time. `/dev/urandom`

numbers, we can combine four random bytes to form random integers. This requires some trick with pointers and we show the code below:

```
#include <stdio.h>

int main(void)
{
unsigned int give_random();

printf(“The number returned is %u\n”,give_random_int());

return(0);
}

unsigned int give_random_int()
{
unsigned int *bhabol;
char puchkay[4];
FILE *in;

in = fopen(“/dev/urandom”,”rt”);
bhabol = (unsigned int*)puchkay; /* fill bhabol with the
address of puchkay[0] */
fscanf(in,”%c%c%c%c”,&puchkay[0],&puchkay[1],&puchkay[2],&puchkay[3]);
return(*bhabol); /* four consecutive bytes interpreted as an
unsigned int */
}
```

Note what we have done. We have used a type cast[46] in front of the address of `puchkay[0]` to change the type to a pointer to an unsigned integer.

manages to generate random bytes even when new events are not occurring; the random bytes it gives are not that *good*, but they are good enough (even more than that) for us!

[46] A type cast is a way of passing a data through a filter (actually a temporaty variable) so that its type is changed. Thus:

```
int a = 2, b = 3;
float x;
x = a/b;
```

will fill up x with a zero because integer division will be used, but:

```
x = ((float)a)/b;
```

will first load the value of `a` into a temporary float variable and divide that by `b`; so we will get `0.6666667` etc. For type casting, you have to put the type you want, enclosed in brackets, before the variable whose type you want to filter.

The same address is now assigned to `bhabol` (the compiler would have given an warning if the type was not changed - a character address on the right and a pointer to `unsigned int` on the left) so that what appears to `puchkay` as the beginning of character array appears to `bhabol` as the beginning of a sequence of four bytes which represent an unsigned integer. `*bhabol` returns the value of that unsigned integer.

This is a way of generating *very good* random numbers, but is a rather slow method because file open and read operations are involved - OS calls are always slow.

### 1.3.12 Relatively advanced topics

#### 1.3.12.1 Bit manipulations

#### 1.3.12.2 Dynamic memory allocation with `malloc`

#### 1.3.12.3 Command line arguments

#### 1.3.12.4 Varargs

# Index