C Programming

Chapter 1: INTRODUCTION

Introduction

History

C was created by Dennis Ritchie and B.W. Kernighan at Bell Labs in the 1972 for a specific purpose to design UNIX operating system. The C language is so named because its predecessor was called B which was developed by Ken Thompson of Bells Labs.

In 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988

What is C

C is a **computer programming language** used to create a list of instructions to be followed by the computer. C is one of thousands of programming languages currently in use.

This high level programming language has proved to be a powerful and flexible language that can be used for a variety of applications: database systems, graphics packages, spreadsheet, word processors, business programs, scientific and engineering applications. C is a particularly popular language for personal computer programmers because it is relatively small - it requires less memory than other languages.

Features (advantages) of C

1. **C is portable**
   This means that a c program written for one computer system (an IBM PC, for example) can be compiled and run on another system with a little or no modification. Machine language or assembly language varies from computer to computer and hence, program written in these languages are not portable. But in the case of High level languages such as C, PASCAL, FORTRAN, the language remains the same, irrespective of the environment. Programs written in these languages can be executed in different computers and operating systems if compilers are available for these systems. A program written with the Microsoft Windows operating system can be moved to a machine running Linux with little or no modification.

2. **C is Powerful and Flexible**
   What you can accomplish is limited only by your imagination. The language itself places no constraints on you. C is used for project as diverse as operating system, word processors, spreadsheet and even compiler of other language.

C combines the convenience and portable nature of a high-level language with the flexibility of low-level language.

3. C has few keywords

It contains only a handful of terms called **Keywords** (reserved words), which serve as the base on which the language's functionality is built.

4. Wide acceptability (Popularity)

C is a language known by the majority of programmers around the world. C is a popular language preferred by professional programmers, it has a powerful compiler.

Some terms

Development Software: programs used to create application software. Programs are written in a programming languages.

Language :is a set of characters, symbols and rules of their assembling for communication purposes. Communication is assured between different individuals with the help of language (spoken, written, sign, symbols).

There are 2 types of languages:

-**Natural language**: spoken by human being or animals

e.g: Swahili, English, French,...

-**Artificial language** :used to simplify communication especially in computer programming.

There're similarities between those languages. Each language has its own grammatical rules which must be obeyed in order to write valid programs just as natural language has its own grammatical rules for forming sentences.

Program: a series of instructions organised in algorithms to accomplish a specific task.

Computer Programming Language: is one (language) the programmer uses to express his/her solution to a given problem so that can be understood by the machine (computer).

They are divided into 2 categories:

-**Low level language**: is a programming language that is machine dependent, it runs on only one particular type of computer. There are 2 types:

-**Machine language:** known as the first generation of programming languages, is the only language the computer recognizes. Machine language instructions uses a series of binary (0's and 1's) or a combination of number and letters that represents binary digits.

-**Assembly language:** the second generation of programming languages, a programmer writes instructions using symbolic codes, these are meaningful abbreviations and codes.

e.g: A- for addition
C- for compare
L- for load
M- for multiply

-High-level language

A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages. In contrast, assembly languages are considered low-level because they are very close to machine languages.

The main advantage of high-level languages over low-level languages is that they are easier to read, write, and maintain. Ultimately, programs written in a high-level language must be translated into machine language by a compiler or interpreter. The first high-level programming languages were designed in the 1950s.

High level language includes:

- procedural languages
- visual programming languages
- object oriented languages
- non-procedural languages
- web page languages
- multimedia languages, ...

Procedural language: the disadvantages of machine and assembly languages led to the development of procedural languages.

In a procedural language, the programmer assigns a name(s) to a sequence of program instructions called a procedure that tells the computer what accomplish and how to do it.

With a procedural language often called a third generation language(3GL) a programmer uses a series of English like words to write instructions.

e.g: ADD stands for addition

PRINT means display

Many 3GLs also use arithmetic operator such as + for addition.

These English words and arithmetic symbols simplify the program development process for the programmer. Other 3GLs are BASIC,COBOL,C,....

Interpreter

A program that executes instructions written in a high-level language. There are two ways to run programs written in a high-level language. The most common is to compile the program; the other method is to pass the program through an interpreter.

Compiler

A program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an interpreter, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Source code: a series of statements or commands that are used to instruct the computer to perform your desired task. To get from source code to machine language, the source code must be translated by a compiler.

Object code: it is often the same or similar to computer machine's language**.** The code

produced by a compiler. Programmers write programs in a form called source code. The source code consists of instructions in a particular language, like C or FORTRAN. Computers, however, can only execute instructions written in a low-level language called machine language.

Stages in program Development

There are certain structures approaches that one keeps in mind while creating a computer program:

1 .Define or understand the problem: first of all, you must understand the problem, if you don't know the problem, you can't find the solution.

2 Analyse the problem: once you know what the problem is, you can analyse it and make a plan to resolve it.

3. Develop an algorithm and flowchart: this is a process whereby a set of instructions are used to produce a solution to a given problem

Any computer program contain instructions in 3 main categories:

-input instructions: used for supplying data to a program inside the computer.

-processing instructions: used for manipulating data inside the computer like addition, multiplication ,subtraction,...

-output instructions: used for getting out information

4. Writing the computer's code: following the algorithm and flowchart. This is the next step where you write the codes for the program to make it work.

5. Compiling and Debugging the program: once the program coding is completed, you compile your program means you translate the source code to object code and if there are errors, you debug them.

Debugging means to correct or remove errors (bugs) if there are in the program.

Programming errors come in 3 varieties

a) Compile (syntax) error: you get it when you've broken the rules of computer programming language.

e.g: spelling printf as prinntf or printif

You also receive a compiler error, if accidentally use the wrong punctuation or place a punctuation in the wrong place.

b) Run-time error: it can be caused by attempting to do impossible arithmetic operations, such calculating non-numeric data, dividing a number by zero, or find the square root of a negative number.

c) Logic error: with this, your application runs but produces incorrect result. Perhaps the results of calculation are incorrect or the wrong text appears, or the text is ok but appears in the wrong location.

Note that errors are unavoidable part of program development. Your C compiler detects errors in your source code and displays an error message, giving both the nature and the location of

the error. Using this information, you can edit your source code to correct the error. Remember, however that the compiler can't always accurately report the nature and location of an error, sometimes, you need to use your knowledge of C to track down exactly what is causing a given error message.

6 .Running the program: to run an application means to execute it and check, if using some data, it is working with the correctness of the program.

7. Implementation and Documentation: when those steps are achieved, you could implement your application. You may add the explanation on how program works and how to use it.

Program Development Cycle

Developing a program in a compiled language like C requires at least four steps:

1. **Editing (or writing) the program**: You write a computer program with words and symbols that are understandable to human beings. This is the editing part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file with a .c extension.
2. **Compiling it**: You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit (for example, the Pentium 4 microprocessor). This process produces an intermediate object file - with the extension `.obj`, the `.obj` stands for Object

3. Linking it : Many compiled languages come with library routines which can be added to your program. Theses routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (`stdio.h`) so even the most basic program will require a library function. After linking the file extension is `.exe` which are executable files.

3. **Executing** it : Thus the text editor produces `.c` source files, which go to the compiler, which produces `.obj` object files, which go to the linker, which produces `.exe` executable file. You can then run `.exe` files as you can other applications

Step 1 : Use an editor to write your source code(the program). By tradition C source code files have the extension .c That means that you have to save the source code with .c extension. ex: Hello.c

Step 2: Compile the program using a compiler. If the compiler doesn't find any errors in the program, it produces an abject file. The compiler produces object files with an .obj extension and the same name as the source code file(for example Hello.c to Hello.obj). If the compiler finds errors, it responds(reports) them, you must return to step 1 to make the corrections in your source code.

Step 3: Link the program with a Linker. If no errors occur, the Linker produces an executable program with an .exe extension and the same name as the object file(for example Hello.obj is linked to create Hello.exe).

Steps 4 Execute the program to see the results. You should test to determine whether it

functions properly, if no start again with step 1 and make modification and addition to your source code.

C's Character Set

C does not use, nor requires the use of, every character found on a modern computer keyboard. The only characters required by the C Programming Language are as follows:

A - Z
a -z
0 - 9
space . , : ; ' $ "
% & ! _ {} [] () < > | + - / * =

The general form of a C program is as follows :

```
pre-processor directives
global declarations
main()
{
   local variables to function main ;
   statements associated with function main ;
}
f1()
{
   local variables to function 1 ;
   statements associated with function 1 ;
}
f2()
{
   local variables to function f2 ;
   statements associated with function 2 ;
}
.
.
.
etc
```

Note the use of the bracket set () and {}. () are used in conjunction with function names whereas {} are used as to delimit the C statements that are associated with that function.

The Simplest C Program

Let's walk through this program and start to see what the different lines are doing

```
#include <stdio.h>
 main()
{
     printf("Hello!\n");
}
```

Program Output:

Hello!

This C program starts with **#include <stdio.h>.** This line includes the "standard I/O library" into your program. The standard I/O library lets you read input from the keyboard (called "standard in"), write output to the screen (called "standard out"), process text files stored on the disk and so on. It is an extremely useful library. C has a large number of standard libraries like stdio, including string, time and math libraries. A library is simply a package of code that someone else has written to make your life easier

-The line **main()** declares the main function. Every C program must have a function named main somewhere in the code. At run time, program execution starts at the first line of the main function.

In C, the **{** and **}** symbols mark the beginning and end of a block of code. In this case, the block of code making up the main function contains two lines.

-The **printf** statement in C allows you to send output to standard out (for us, the screen). The portion in quotes is called the format string and describes how the data is to be formatted when printed. The format string can contain string literal such as "This is output from my first program!," symbols for carriage returns (\n), and operators as placeholders for variables (see below). The **`printf`** function does what its name suggest it does: it prints, on the screen, whatever you tell it to. The **`"\n"`** is a special symbols that forces a new line on the screen.

You note that the semicolon marks the end of an instruction (statement)

Second Program

Write this program:

```
#include <stdio.h>
 main()
{
     printf("Hello, software\n");
}
```

Program Output:

Hello, software

This program, in fact, consists of a single piece or chunk of executable code known as a function . Later on, we will see programs consisting of many functions. All C programs must include a function with the name **main** , execution of C programs always starts with the execution of the function **main** , if it is missing the program cannot run and most compilers will not be able to finish the translation process properly without a function called main .

It is important to note that the required function is called main not MAIN . In the C programming language the case of letters is always significant unlike many older programming languages.

In the sample **hello software** program **main** appears on line 1. This is not essential, program execution starts at **main** not at the first line of the source. However, it is conventional in C programs, to put **main** at, or near, the start of the program.

The round brackets, or parentheses as they are known in computer programming, on line 1 are

essential to tell the compiler that when we wrote **main** we were introducing or defining a function rather than something else. You might expect the compiler to work out this sort of thing for itself but it does make the compiler writer's task much easier if this extra clue is available to help in the translation of the program.
On lines 2 and 4 you will see curly brackets or braces. These serve to enclose the body of the function **main**. They must be present in matched pairs.

Line 3, the word **printf** clearly has something to do with printing. Here, of course, it is actually causing something to be displayed to the screen, but the word print has been used in this context since the time before screens were generally used for computer output. The word **printf** was probably derived from **print formatted** but that doesn't matter very much, all you need to know is that this is the way to get something onto the screen.

In computer programming**, printf** is a library function. This means that the actual instructions for displaying on the screen are copied into your program from a library of already compiled useful functions.
Line 3 is a statement. **A statement** tells the computer to do something. In computer programming, we say that a statement is executed. This simple example consists of the use of a library function to do something. In the C programming language a simple use of a function (library or otherwise) is, technically, an expression, i.e. something that has a value. In computer programming we say that an expression is evaluated. To convert an expression into a statement a semi-colon must follow the expression, this will be seen at the end of line 3.
You can include as many \n's as you like within a string enabling multi-line output to be produced with a single use of the **printf()** function.

Here's an example

```
#include <stdio.h>
main()
{
    printf("Hello,\n software\n development\n");
}
```

There are several other characters that cannot conveniently be included in a string but which can be included using the \ notation.
The complete list is:

\a	“bell” – i.e a beep
\b	Backspace
\f	Form Feed (new page)
\n	New Line (line feed)
\t	Tab
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark

Add Comments to a Program

A comment is a note to yourself (or others) that you put into your source code. All comments are ignored by the compiler. They exist solely for your benefit. Comments are used primarily to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it. You can also use a comment to temporarily remove a line of code. Simply surround the line(s) with the comment symbols.
In C, the start of a comment is signaled by the **/*** character pair. A comment is ended by ***/**.
For example, this is a syntactically correct C comment:

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

Comments can extend over several lines and can go anywhere except in the middle of any C keyword, function name or variable name. In C you can't have one comment within another comment. That is comments may not be nested. Lets now look at our first program one last time but this time with comments:

```
main() /* main function heading */
{
   printf("\n Hello, World! \n");  /* Display message on */
}                                   /* the screen */
```

Chapter 2: DATA TYPES

The first thing you need to know is that you can create variables to store values in.
A variable is just a named storage area in the computer's memory that can hold a single value (numeric or character).
It demands that you declare the name of each variable that you are going to use and its type, or before you actually try to do anything with it.
There are four basic data types associated with variables:

int - integer: a whole number.
float - floating point value: ie a number with a fractional part.
double - a double-precision floating point value.
char - a single character.

Integer Number Variables

The first type of variable we need to know about is of class type **`int`** - short for integer. The simplest type of numeric data that a programmer can define in a C program is integer data An **`int`** variable can store a value in the range -32768 to +32767. You can think of it as a largish positive or negative whole number: no fractional part is allowed. To declare an **`int`** you use the instruction:

```
int variable name;
```

For example:

```
int a;
```

declares that you want to create an **`int`** variable called **`a`**.
To assign a value to our integer variable we would use the following C statement:

```
a=10;
```

The C programming language uses the "=" character for assignment. A statement of the form **`a=10;`** should be interpreted as **take the numerical value 10 and store it in a memory location associated with the integer variable a**.
The "=" character should not be seen as an equality.

Decimal Number Variables

As described above, an integer variable has no fractional part. Integer variables tend to be used for counting, whereas real numbers are used in arithmetic. C uses one of two keywords to declare a variable that is to be associated with a decimal number: **`float`** and **`double`**. They are each offer a different level of precision as outlined below.

float
A float, or floating point, number has about seven digits of precision and a range of about 1.E-36 to 1.E+36. A float takes four bytes to store.
double
A double, or double precision, number has about 13 digits of precision and a range of about 1.E-303 to 1.E+303. A double takes eight bytes to store. It is an exponential number, that is a number follow by the letter e (or E), and an integer corresponding to the power of 10 (signed or not, that is preceded by one + or of one-) 2.75e-2 35.8E+10.25e-2 .

For example:

```
float total;
double sum;
```

To assign a numerical value to our floating point and double precision variables we would use the following C statement:

```
total=0.0;
sum=12.50;
```

Character Variables

C only has a concept of numbers and characters

To declare a variable of type character we use the keyword **`char`**. - A single character stored in one byte.

For example:

```
char c;
```

To assign, or store, a character value in a **`char`** data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if **`c`** is a **`char`** variable you can store the letter **A** in it using the following C statement:

```
c='A'
```

Notice that you can only store a single character in a **`char`** variable.

Assignment Statement

Once you've declared a variable you can use it, but not until it has been declared - attempts to use a variable that has not been defined will cause a compiler error. Using a variable means storing something in it. You can store a value in a variable using:

```
name = value;
```

For example:

```
a=10;
```

stores the value **`10`** in the **`int`** variable **`a`**

Consider four very simple mathematical operations: add, subtract, multiply and divide. Let us see how C would use these operations on two float variables **`a`** and **`b`**.

add

```
a+b
```

subtract

```
a-b
```

multiply

```
a*b
```

divide

```
a/b
```

Note that we have used the following characters from C's character set:

```
+     for add
-     for subtract
*     for multiply
/     for divide
```

BE CAREFUL WITH ARITHMETIC!!! What is the answer to this simple calculation?

```
a=10/3
```

The answer depends upon how **`a`** was declared. If it was declared as type **`int`** the answer will

be 3; if **a** is of type **float** then the answer will be 3.333. It is left as an exercise to the reader to find out the answer for **a** of type **char**.
Two points to note from the above calculation:
C ignores fractions when doing integer division.
when doing **float** calculations integers will be converted into **float**.

Arithmetic Ordering

Consider the following calculation:

```
a=10.0 + 2.0 * 5.0 - 6.0 / 2.0
```

All mathematical operations form a hierarchy which is shown here. In the above calculation the multiplication and division parts will be evaluated first and then the addition and subtraction parts. This gives an answer of 17.

Note: To avoid confusion use brackets. The following are two different calculations:

```
a=10.0 + (2.0 * 5.0) - (6.0 / 2.0)
a=(10.0 + 2.0) * (5.0 - 6.0) / 2.0
```

You can freely mix **int**, **float** and **double** variables in expressions

Before you can use a variable you have to declare it. You can declare any number of variables of the same type with a single statement. For example:

```
int a, b, c;
```

Input and Output Functions

Input function: **scanf**

Output function: **printf**

There are a number of different C input commands, the most useful of which is the **scanf** command. To read a single integer value into the variable called **a** you would use:

```
scanf("%d",&a);
```

When the program reaches the **scanf** statement it pauses to give the user time to type something on the keyboard and continues only when users press <Enter>, or <Return>, to signal that he, or she, has finished entering the value. Then the program continues with the new value stored in **a**. In this way, each time the program is run the user gets a chance to type in a different value to the variable and the program also gets the chance to produce a different result!

Using the **printf** function, the one we have already used to print "Hello", to print the value currently being stored in a variable. To display the value stored in the variable **a** you would use:

```
printf("The value stored in a is %d",a);
```

The **%d**, both in the case of **scanf** and **printf**, simply lets the compiler know that the value being read in, or printed out, is a decimal integer - that is, a few digits but no decimal point.

The output manipulator
Initializing the variables
Example

```
#include<stdio.h>
main()
{
int a,b,c;
a=5;   /* variable initialisation */
b=10;
c=a+b;
printf("%d +%d =% d\n",a,b,c);
}
```

The Input/Output manipulators : printf and scanf

The previous program is good, but it would be better if it read in the values 5 and 10 from the user instead of using constants. Try this program instead:

```
#include <stdio.h>

 main()
{
     int a, b, c;
     printf("Enter the first value:\n");
     scanf("%d", &a);
     printf("Enter the second value:\n");
     scanf("%d", &b);
     c = a + b;
     printf("%d + %d = %d\n", a, b, c);
}
```

Program output:
Enter the first value:10
Enter the second value:15
10+15=25

Here's how this program works when you execute it: Make the changes, then compile and run the program to make sure it works. Note that scanf uses the same sort of format string as printf. Also note the & in front of a and b. This is the **address operator** in C: It returns the address of the variable You must use the & operator in scanf on any variable of type char, int, or float. If you leave out the & operator, you will get an error when you run the program. Try it so that you can see what that sort of run-time error looks like.

Let's look at some variations to understand printf completely. Here is the simplest printf statement:

```
printf("Hello");
```

This call to printf has a format string that tells printf to send the word "Hello" to standard out. Contrast it with this:

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

The difference between the two is that the second version sends the word "Hello" followed by a carriage return to standard out.

The following line shows how to output the value of a variable using printf.

```
printf("%d", b);
```

The **%d** is a placeholder that will be replaced by the value of the variable **b** when the printf statement is executed. Often, you will want to embed the value within some other words. One way to accomplish that is like this:

```
printf("The temperature is ");
printf("%d", b);
printf(" degrees\n");
```

An easier way is to say this:

```
printf("The temperature is %d degrees\n", b);
```

You can also use multiple %d placeholders in one printf statement:

```
printf("%d + %d = %d\n", a, b, c);
```

In the printf statement, it is extremely important that the number of **operators** in the format string corresponds exactly with the number and type of the variables following it. For example, if the format string contains three %d operators, then it must be followed by exactly three parameters and they must have the same types in the same order as those specified by the operators.

Here is a picture(board) giving the types of data in language C:

Data type	Description	Size (in bytes)	Range
char	Character	1	-128 to 127
unsigned char	unsigned Character	1	0 to 255
short int	short integer	2	-32768 to 32767
unsigned short int	unsigned short integer	2	0 to 65535
int	integer	2 (on processor 16 bits)	-32768 to 32767
		4 (on processor 32 bits)	-2147483648 to 2147483647
unsigned int	unsigned integer	2 (on processor 16 bits)	0 to 65535
		4 (on processor 32 bits)	0 to 4294967295
long int	long integer	4	-2 147 483 648 to 2 147 483 647
unsigned long int	unsigned long integer	2	0 to 4 294 967 295
float	float	4	$3.4*10^{-38}$ to $3.4*10^{38}$
double	double float	8	$1.7*10^{-308}$ to $1.7*10^{308}$
long double	double long float	10	$3.4*10^{-4932}$ to $3.4*10^{4932}$

In C, variable names must adhere to the following rules:

- The name can contain letters, digits, and the underscore(_)
- The first character of the name must be a letter. The underscore is also legal first character, but its use is not recommended a the beginning of the name
- Case matters(that is uppercase and lowercase letters). C is case-sensitive, thus the names count and Count refer to two different variables.
- C keyword can not be used as a variable name. A keyword (reserved word) is a word that is the part of the C language.
- only the first 31 characters of a variables name are significant

Here there are keywords of C

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

A variable declares itself in the following way:

Type Variable_Name ;

Either if there are several variables of the same type:

Type Variable_Name1, Variable_Name2;

Assignment of a data in a variable

To store a data in a variable which we initialized, it is necessary to make an affectation (appointment), that is to clarify the datum which is going to be stored in the place memory (report) which was reserved during the initialization.

For it we use the operator of affectation (appointment or assignment) **"="**:

Variable_Name= data;

To store the character B in the variable which we called **Character**, it will be necessary to write:

Character = ' B ';

What means storing the value ASCII of "**B**" in the variable named "**Character**". It is very evident that it is beforehand necessary to have declared the variable by allocating to it the type

Char;

Char Character;

Initialization of a variable

The statement(declaration) of a variable is only "reserving" a place memory(report) where to store the variable.
We can thus allocate an initial value to the variable during its statement(declaration), we speak then about initialization:

type Variable_Name = data;

For example: **float Toto = 125.36;**

You can print all of the normal C types with printf by using different placeholders:

- **int** (integer values) uses **%d**
- **float** (floating point values) uses **%f**
- Double(double floating point values) uses %e
- **char** (single character values) uses **%c**
- **character strings** (arrays of characters, discussed later) use **%s**

The intput I/O manipulator: Scanf

The **scanf function allows you to accept input from standard in**, which for us is generally the keyboard. The scanf function can do a lot of different things, but it is generally unreliable unless used in the simplest ways. It is unreliable because it does not handle human errors very well. But for simple programs it is good enough and easy-to-use.

The simplest application of **scanf** looks like this:

```
scanf("%d", &b);
```

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b.

The scanf function uses the same placeholders as printf i.e **Formatters for scanf()**

The following characters, after the % character, in a scanf argument, have the following effect.

Modifer	**Meaning**
d	read a decimal integer
o	read an octal value
x	read a hexadecimal value
h	read a short integer
l	read a long integer

f	read a float value
e	read a double value
c	read a single character
s	read a sequence of characters, stop reading when an enter key or whitespace character [tab or space]

There is a function in C which allows the programmer to accept input from a keyboard. The following program illustrates the use of this function,

```
#include <stdio.h>
main()      /* program which introduces keyboard input */
{
        int   number;
        printf("Type in a number \n");
        scanf("%d", &number);
        printf("The number you typed was %d\n", number);
}
```

Sample Program Output

```
Type in a number
23
The number you typed was 23
```

An integer called number is defined. A prompt to enter in a number is then printed using the statement

```
printf("Type in a number \n:");
```

The scanf routine, which accepts the response, has two arguments. The first ("%d") specifies what type of data type is expected (ie char, int, or float).

The second argument (&number) specifies the variable into which the typed response will be placed. In this case the response will be placed into the memory location associated with the variable number.

This explains the special significance of the & character (which means the address of).

Sample program illustrating use of scanf() to read integers, characters and floats

```
#include < stdio.h >
main()
{
        int sum;
        char letter;
        float money;
        printf("Please enter an integer value ");
        scanf("%d", &sum );
        printf("Please enter a character ");
        /* the leading space before the %c ignores space characters in the input */
        scanf("   %c", &letter );
```

```
        printf("Please enter a float variable ");
        scanf("%f", &money );

        printf("\nThe variables you entered were\n");
        printf("value of sum = %d\n", sum );
        printf("value of letter = %c\n", letter );
        printf("value of money = %f\n", money );
}
```

Sample Program Output

```
Please enter an integer value
34
Please enter a character
W
Please enter a float variable
32.3
The variables you entered were
value of sum = 34
value of letter = W
value of money = 32.300000
```

N.B: Between the % that introduces the conversion specification in the printf() format string and the f ,that terminates the conversion there will usually be a **field specification** of the form **w.d**
where w specifies the overall **field width** and d is the **precision** specification which tells printf() how many digits to print after the decimal point. If the precision is not specified then a default of 6 is used.

CONSTANTS

Like a Variable, a constant is data storage location used by your program, Unlike variable, the value stored in constant can't be changed during the program execution. C has two types of constants, each with each own specific uses:

- Literal Constants
- Symbolic Constants

Literal Constants

A literal constant is a value that is typed directly into the source code whenever its needed.
Here are two examples:

```
Int count=20,
Float tax_rate=0.20
```

The 20 and 0.20 are literal constants. The preceding statement store these values in the

variable count and tax_rate. The presence or absence of the decimal point distinguishes floating point constants from inter constants.

Symbolic constants

A symbolic constant is a constant that is represented by a name(symbol) in your program. Like literal constant, a symbolic constant can't change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it first defined.

Defining a symbolic constant

C has two methods for defining a symbolic constant:
The #define directive is used as follows:

The first one is in this format

#define CONSTANTNAME literal
ex: #define PI 3.14159

By convention the names of the symbolic constant are uppercase, this make them easy to distinguish from variables names which by convention are lowercase.

The second method is like that:
const int count=20;
const float pi=3.12159;

Exercises:

1. Write a c program that displays your name on the screen.
2. Write a c program that displays days of week on the screen
3. Write a program to convert from Celsius degree (^{0}C) to Fahrenheit degrees (^{0}F)

4. Write a c program to read the name and marks of a student for 3 subjects (Math, chemistry and Computer) from keyboard. The program should calculate the total and average marks for the student.
5. Write a c program that calculate the area of a circle using constants.
6. Write a c program that convert from Rwandese money to dollars
7. Write a c program that read age, sex, height of a given student from the keyboard and display the output on the secreen.

Chapter 3 EXPRESSIONS AND OPERATORS

One reason for the power of C is its wide range of useful operators. An **operator** is a function which is applied to values to give a result.
You should be familiar with operators such as +, -, /. Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.
Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

1. Arithmetic Operators

The symbols of the arithmetic operators are:

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6
Subtraction	-	sum = sum -2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

Here are some arithmetic expressions used within assignment statements.

```
velocity = distance / time;

force = mass * acceleration;

count = count + 1;
```

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

These operations are usually very efficient. They can be combined with another expression.

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - - to ensure that your programs stay readable.

Another shorthand notation is listed below

These are simple to read and use.

2. The Relational Operators

These allow the comparison of two or more variables.

Operator	**Meaning**
==	equal to
!=	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

3. Logical Operators (AND, NOT, OR)

Combining more than one condition

These allow the testing of more than one condition as part of selection statements.

The symbols are

Logical AND &&

Logical and requires all conditions to evaluate as TRUE (non-zero).

Logical OR ||

Logical or will be executed if any ONE of the conditions is TRUE (non-zero).

Logical NOT !

Logical not negates (changes from TRUE to FALSE, vsvs) a condition.

4.Conditional operator

This conditional expression operator takes 3 operators. The 2 symbols used to denote this operator are **?** and the **:**.The first operand is placed before ?, the second one between the ? and the : and the third after:

The general format is,

Conditional **?** expression 1**:** expression 2

If the result of condition is TRUE ,expression 1 is evaluated and the result of the evaluation becomes the result of the operation.

If the condition is FALSE, the expression2 is evaluated and its result becomes the result of the operation

Examples:

1)

```
s=(x<0)?-1:x*x;
```

if x is less than zero then s=-1

if x is greater than zero then s= x*x

Evaluation of Expressions

The expression `x+y+z` illustrates an important point. Since the operator "+" is a binary operator this expression has to be evaluated in 2 steps. These could be either

Evaluate the expression x+y
Add z to the result of the previous evaluation

Or

Evaluate the expression y+z
Add x to the result of the previous evaluation

The computer to do it either way and it clearly makes no difference to the result.

However for the expression `x-y+z` it clearly does make a difference as the following quick calculation shows.
First assume x has the value 1, y has the value 2 and z has the value 3. Evaluating x-y and then adding z to the result gives the value +2 whereas evaluating y+z and then subtracting the result from x gives the value -4.

The "+" and "-" operators **group** or **associate** left-to-right. This means that an expression such as x-y+z is evaluated from left to right corresponding to the first alternative described above. If we weren't certain what this meant, a quick program confirms that our compiler writer interpreted it the same way as we did.

```
# include<stdio.h>
main()
{
     int x=1,y=2,z=3;
     printf("Value of \"%d-%d+%d\" is %d\n",   x,y,z,x-y+z);
    printf("Value of \"%d+%d-%d\" is %d\n", x,y,z,x+y-z);
}
```

Giving the following output
Value of "1-2+3" is 2
Value of "1+2-3" is 0

Suppose now that we did want to calculate y+z first and then subtract the result from x. This problem can be solved by using an extra piece of notation and writing the expression as `x-(y+z)` The **parentheses** enclose an expression and expressions enclosed within parentheses are evaluated before other expressions according to the standard. Further expressions contained within parentheses can be enclosed within parentheses to an arbitrary depth, this is called **expression nesting.** Expressions enclosed within parentheses are sometimes called **sub-expressions** but this isn't really very helpful as they are proper expressions in their own right.

Again a simple programming example sufficient to convince ourselves that parentheses work as advertised.

```
#include<stdio.h>
main()
{
int       x=1,y=2,z=3;
printf("Value of \"%d-%d+%d\" is %d\n", x,y,z,x-y+z);
printf("value of \"%d+%d-%d\" is %d\n",x,y,z,x+y-z);
getch();
}
```

There is an important difference between the "+" and the "-" operators. The "+" operator is a **commutative** operator whereas - is not. The adjective commutative applied to a binary operator means that it doesn't matter which order it takes its operators in.
In other words

Value1 operator value2 has exactly the same value as Value2 operator value1

This is clearly true for "+" and equally clearly false for "-".

The expressions involving only one of the commutative and associative operators can be evaluated in any order.
The only associative and commutative operators we have met so far are "+" and "*".

Value1 operator (value2 operator value3)
(Value1 operator value2)operator value3
Value1 operator value2 operator value3

Operator Precedence

The expressions `x+y*z` and `x*y+z` suffer from the same potential ambiguity as `x+y-z` and `x-y+z` however the problem is handled in a different way. The normal mathematical expectation is that multiplication is performed before addition. There are various ways of saying this, we could say that the "*" operator **binds more tightly** or we could say, and will say, that the "*" operator has a higher **precedence** than the "+" operator.

Again a simple programming example confirms this point.

```
# include<stdio.h>
main()
{
int       x=2,y=7,z=5;
clrscr() ;
printf("The value of \"%d*%d+%d\" is %d\n", x,y,z,x*y+z);
printf("The value of \"%d+%d*%d\" is %d\n",z,x,y,z+x*y);
getch();
```

```
}
```

The output
The value of "2*7+5" is 19
The value of "5+2*7" is 19

Of course, we could use parentheses if we actually wanted addition performed before multiplication as the following example shows.

```
#include<stdio.h>
main()
{
int       x=2,y=7,z=5;
clrscr() ;
printf("The value of \"%d*(%d+%d)\" is %d\n", x,y,z,x*(y+z));
printf("The value of \"(%d+%d)*%d\" is %d\n", z,x,y,(z+x)*y);
getch();
}
```

resulting in the output

The value of "2*(7+5)" is 24
The value of "(5+2)*7" is 49

Exercises:

1. Find the value assign to the variable on the left of the assignment operator +. Work each problem independently of the others. Assume the following declarations:

```
int   i=2,
        j=3,
        k=4,
result;
```

1. result = 5*i-j;
2. result = 5*(i-j);
3. result = 33/i*j;
4. result = 33/(i*j);
5. result = 33% i*j;
6. result   = i-j+k;
7. result   = i-(j+k);
8. result   = 17% k-i*j+6;
9. result   = 17%(k-i)*j+6;
10. result   = i*(7+(j+3)/2)-k;
11. j = j+k;

12. i = i*2;
13. i = i%3;
14. i = i/j+k;
15. k = k- 5;
16. k = (i+j) * k / 4;

2. /*A program to read in two numbers and print their sum*/

```
#include <stdio.h>
main()
{
        int     x,y;    /* places to store numbers */
        printf("Enter x ");
        scanf("%d",&x);
        printf("Enter y ");
        scanf("%d",&y);
        printf("The sum of x and y is %d\n",x+y);
        getch();
}
```

3. /*Another sum of two numbers

```
#include<stdio.h>
main()
{
int     n1;
int     n2;
printf("Enter two numbers ");
scanf("%d%d",&n1,&n2);
printf("The sum is %d\n",n1+n2);
getch();
}
```

Summary:

- Every C program contains a function main() that controls execution of the program
- Every C program statement must end in a semicolon
- Use printf to display strings and variable values.
- Always code the preprocessor directive # include <stdio.h> in all your programs
- An identifier (variable)can consist of up to 31 characters.
- A variable is a named location in computer memory that stores a particular type of data.
- A variable declaration must begin with the variable's data type.
- The integer are int, unsigned, long, unsigned long, short, and unsigned short.
- Use scanf to obtain a datum from the input stream(the keyboard) and place in to a variable.

- The main arithmetic operators are +,-,*,/,and %.
- The arithmetic operators follow the usual precedence rules.
- Use const to declare defined constants.
- A statement is a complete direction instructing the computer to carry out some task.
- A group of one or more statement enclosed within braces is called a **Block**.

Exercise:
Write a c program that reads an integer number of three digits and print it in reverse order

Chapter 4 DECISION CONTROL STATEMENT

In C, one of several possible actions will be carried out depending upon the outcome of the logical test which is called branching.

Some portion of the program has to be executed several number of times or until a particular condition is being satisfied. This is called looping.

IF ELSE ELSE IF statement

General syntax:

if, else if, ….., else

```
if(expression)
{
statements;
}
else if(expression)
{
statements;
}
……..
else(expression)
{
statements;
}
```

Here is a simple C program demonstrating an if statement:

```
#include <stdio.h>
main()
{
    int b;
    printf("Enter a value:");
    scanf("%d", &b);
    if (b < 0)
        printf("The value is negative\n");
}
```

This program accepts a number from the user. It then tests the number using an if statement to see if it is less than 0. If it is, the program prints a message. Otherwise, the program is silent. The **(b < 0)** portion of the program is the **Boolean expression**. C evaluates this expression to decide whether or not to print the message. If the Boolean expression evaluates to **True**, then C executes the single line immediately following the if statement (or a block of lines within braces immediately following the if statement). If the Boolean expression is **False**, then C

skips the line or block of lines immediately following the if statement.

Here's slightly more complex example:

```
#include<stdio.h>
Main()
int b;
printf("Enter the value of b:");
scanf("%d",&b);
if(b<0)
{
printf("The value of b is negative \n");
}
else if(b==0)
{
printf("The value of b is zero\n");
}
else
{
printf("the value of b is Positive\n");
}
getch();
}
```

In this example, the **else if** and **else** sections evaluate for zero and positive values as well.

Other examples:

e.g$_1$: program for guessing the birthday of somebody

```
#include <stdio.h>
main()
{
    int myBirthday = 13;
    int guess;
    printf("Please guess the day of my birth, from 1 to 31\n");
    printf("Enter your guess, please ");
    scanf("%d",&guess);
                        /* Test for out of range guesses first */
    if (guess <= 0)
    {
        printf("Months have at least one day, Einstein\n");
    }
    else if (guess > 31)
    {
        printf("Pretty long month, genius\n");
    }
    else if (guess = = myBirthday)
    {
       printf("Incredible, you are correct\n");
    }
    else if (guess < myBirthday)
    {
        printf("Higher, try again\n");
    }
    else                    /* Guess is in range and not greater than or equal */
    {
        printf("Lower, try again\n");
 }
 }
```

e.g$_2$: program that works like a simple calculator

```
#include<stdio.h>
main()
{
float num1,num2;
char op;
clrscr();
printf("Enter the first number:");
scanf("%f",&num1);
printf("\n Enter the operator:");
scanf("%s",&op);
printf("\n Enter the second number:");
scanf("%f",&num2);
```

```
if(op= ='+')
{
printf("%f + %f is %f",num1,num2,num1+num2);
}
else if(op= ='-')
{
printf("%f - %f is %f",num1,num2,num1-num2);
}
else if(op= ='*')
{
printf("%f * %f is %f",num1,num2,num1*num2);
}
else if(op= ='/');
{
printf("%f / %f is %f",num1,num2,num1/num2);
}
                    else
{
printf("/n Invalid operator");
}
getch();
}
```

A PROGRAM THAT CALCULATES ROOTS OF A QUADRATIC EQUATION FOR VARIOUS CASES

```
#include<stdio.h>
#include<math.h>
main()
{
 float a,b,c;
 clrscr();
 printf("ENTER THREE CONSTANTS FOR A QUADRATIC EQUATION\n");
 scanf("%f%f%f",&a,&b,&c);
 if(a==0)
 {
 printf("THIS IS NOT A QUADRATIC EQUATION\n");
 printf("THE VALUE OF X =%f",(-c/b));
 }
 else if(((b*b)-(4*a*c))==0)
 {
 printf("THE QUADRATIC EQUATION HAS TWO EQUAL ROOTS\n");
 printf("TWO EQUAL ROOTS ARE ROOT1=ROOT2=%f",(-b/(2*a)));
 }
 else if(((b*b)-(4*a*c))>0)
 {
 printf("THE QUADRATIC EQUATION HAS TWO DISTINCT ROOTS\n");
 printf("THE FIRST ROOT IS ROOT1=%f\n",(-b-sqrt(b*b-4*a*c))/(2*a));
```

```
 printf("THE SECOND ROOT IS ROOT2=%f\n",(-b+sqrt(b*b-4*a*c))/(2*a));
 }
 else
 {
 printf("THE EQUATION HAS COMPLEX ROOTS\n");
 printf("THE FIRST COMPLEX ROOT1=%f-j%f\n",-b/(2*a),sqrt(-(b*b-4*a*c))/(2*a));
 printf("THE SECOND COMPLEX ROOT2=%f+j%f\n",-b/(2*a),sqrt(-(b*b-4*a*c))/(2*a));
 }
 getch();
}
```

switch() case:

The switch case statement is a better way of writing a program when a series of if else occurs.

The general format for this is:

```
switch ( expression ) {
                    case   value 1:
                               program statement;
                               program statement;
                               ......
                               break;
                    case   value 2:
                               program statement;
                               program statement;
                               ......
                               break;
                               …….
                               …….

                    case   value n:
                               program statement;
                               .......
                               break;
                    default:
                               .......
                               .......
                               break;
          }
```

The keyword break must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Rules for switch statements

- Values for 'case' must be integer or character constants
- The order of the 'case' statements is unimportant

- The default clause may occur first (convention places it last)
- You cannot use expressions or ranges

e.g$_1$:

```
#include <stdio.h>
main()
{
        int menu, numb1, numb2, total;

        printf("enter in two numbers -->");
        scanf("%d %d", &numb1, &numb2 );
        printf("enter in choice\n");
        printf("1 = addition\n");
        printf("2 = subtraction\n");
        scanf("%d", &menu );
        switch( menu ) {
                case 1: total = numb1 + numb2; break;
                case 2: total = numb1 - numb2; break;
                default: printf("Invalid option selected\n");
        }
        if( menu = = 1 )
                printf("%d plus %d is %d\n", numb1, numb2, total );
        else if( menu = = 2 )
                printf("%d minus %d is %d\n", numb1, numb2, total );
}
```

Sample Program Output

```
enter in two numbers --> 37 23
enter in choice
1=addition
2=subtraction
2
 37 minus 23 is 14
```

The above program can also be written in the below way

USING SWITCH CONTROL STATEMENT

```
#include<stdio.h>
main()
{
 int a,b,x;
 clrscr();
 printf("PLEASE ENTER TWO VARIABLES\n");
 scanf("%d%d",&a,&b);
 printf("CHOOSE YOUR CHOICE IN THE BELOW MENU\n");
 printf("ENTER 1 FOR DOING ADDITION\n");
 printf("ENTER 2 FOR DOING SUBTRACTION\n");
 printf("ENTER 3 FOR DOING MULTIPLICATION\n");
 printf("ENTER 4 FOR DOING DIVISION\n");
 scanf("%d",&x);
 switch(x)
 {
 case 1:printf("THE RESULT IS =%d\n",a+b);
 break;
 case 2:printf("THE RESULT IS =%d\n",a-b);
 break;
 case 3:printf("THE RESULT IS =%d\n",a*b);
 break;
 case 4:printf("THE RESULT IS =%d\n",a/b);
 break;
 default:printf("INVALID CHOICE");
 }
 getch();
        }
```

e.g2: program that reads Euro or Dollar and converts it into Rwandese money

```
#include<stdio.h>
main()
{
int conversion;
float rw,euro,dollar;
clrscr();
printf(" 1: conversion from European to Rwandese\n");
printf(" 2: conversion from American to Rwandese\n");
printf("enter a choice:");
scanf("%d",&conversion);
switch(conversion)
{
case 1:
printf("enter the amount in European francs: ");
```

```
scanf("%f",&euro);
    rw=euro/700;
    printf("the Rwandese amount is   %.4f",rw);
    break;
    case 2:
    printf("enter the amount in American franc: ");
    scanf("%f",&dollar);
    rw=dollar/600;
    printf("the rwandese amount is   %.2f",rw);
    break;
    default:
    printf("invalid choice selected\n");
    }
    getch(); }
```

e.g3: Program that works as a simple calculator using Switch case

```
#include<stdio.h>
void main()
{
float num1,num2;
char op;
clrscr();
printf("Enter the first number:\n");
scanf("%f",&num1);
printf("Enter a valid operator:\n");
scanf("%s",&op);
printf("Enter the second number:\n");
scanf("%f",&num2);
switch(op)
{
case '+':
 printf("%f + %f is %f",num1,num2,num1+num2);
 break;
case '-':
 printf("%f - %f is %f\n",num1,num2,num1-num2);
break;
case '*':
 printf("%f * %f is %f\n",num1,num2,num1*num2);
break;
case '/':
 printf("%f/%f is %f\n",num1,num2,num1/num2);
break;
default:
printf("\nInvalid operator\n");
break;
getch();
```

}

e.g4: program that prints the day of the week depending of the number entered

```
#include<stdio.h>
main()
{
int choice;
clrscr();
printf("please enter the number of the day you want:");
scanf("%d",&choice);
 switch(choice)
 {
 case 1 :
 printf("Monday");
 break;
 case 2:
 printf("Tuesday");
 break;
 case 3:
 printf("Wednsday");
 break;
 case 4:
 printf("Thrusday");
 break;
 case 5:
 printf("Friday");
 break;
 case 6:
 printf("Suturday");
 break;
 case 7:
 printf("Sunday");
 break;
 default:
 printf("invalid number\n");
 }
 getch();
 }
```

Exercise:

1. Write a simple c program the reads an integer number form keyboard and checks if it is odd or even

Chapter 5 LOOPING

We use loop when you want to execute statement several times until a condition is reached.
Generally, loops consist of two parts:

- one or more control expressions which control the execution of the loop.
- body , which is the statement or a set of statement which is executed over and over.

In C programming, we use 3 types of loop:

- for loop
- while loop
- do while loop

for loops

The basic format of the for statement is,

```
for( start_condition; continue_ condition; re-evaluation )
        {
        program statement;
        }
```

Sample program using a for statement

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

int   count;
clrscr();
for(count= 1;count<= 10;count=count+1)
printf("%d",count);
       getch();
}
```

Sample Program Output

1 2 3 4 5 6 7 8 9 10

SUM OF THE FIRST N NUMBERS

```
#include<stdio.h>
void main()
{
 int i,n,sum=0;
 clrscr();
 printf("ENTER THE LIMIT\n");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 sum=sum+i;
 }
 printf("\n THE SUM OF FIRST %d NUMBERS = %d",n,sum);
 getch();
}
```

The program declares an integer variable count. The first part of the for statement for(count = 1;initialises the value of count to 1.
The for loop continues whilst the condition count <= 10; evaluates as TRUE.
As the variable count has just been initialized to 1, this condition is TRUE and so the program statement printf ("%d ", count);is executed, which prints the value of count to the screen, followed by a space character.
Next, the remaining statement of the for is executed count = count + 1); which adds one to the current value of count. Control now passes back to the conditional test, count <= 10;which evaluates as true, so the program statement printf("%d ", count);
is executed.
Count is incremented again, the condition re-evaluated etc, until count reaches a value of 11.
When this occurs, the conditional test count <= 10; evaluates as FALSE, and the for loop terminates, and program control passes to the statement printf("\n");
which prints a newline, and then the program terminates, as there are no more statements left to execute.

N.B: The **for loop** in C is simply a shorthand way of expressing a while statement.

An example of using a for loop to print out characters

```
#include<stdio.h>

void main()
{
char letter;
clrscr();
for( letter = 'A'; letter <= 'E'; letter = letter + 1 )
{
printf("%c ", letter);
```

```
}
getch();

}
```

Sample Program Output

```
A B C D E
```

Other examples:

e.g$_1$: Program that display multiplication table from 1 to 10.

```
#include<stdio.h>
main(){
int i,j;
for(i=1;i<=10;i++){
for(j=1;j<=10;j++){

printf("%d x %d = %d\n",i,j,i*j);
}
        printf("\n\n");
printf("multiplication table of %d\n",i);
printf("\n\n");
}
getch();
}
```

The above program can be writen in the way below.

MULTIPLICATION TABLE

```
#include<stdio.h>
void main()
{
 int i,table,counter=1;
 clrscr();
 printf("ENTER THE MULTIPLICATION TABLE \n");
 scanf("%d",&table);
 printf("ENTER THE COUNTER VALUE\n");
 scanf("%d",&counter);
 printf("THE REQUIRED TABLE IS SHOWN BELOW:\n");
 for(i=0;i<=counter;i++)
 {
  printf("\n%d * %d = %d",table,i,i*table);
 }
 getch();
}
```

e.g$_2$: Program that reads an integer number from keyboard and calculates its factorial

```
#include<stdio.h>
void main()
{
int i,n;
long int fact=1;
clrscr();
printf("Enter a number:\n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
fact=fact*i;
}
printf("%d!=%li",n, fact);
getch();
}
```

Nested loop

Sometime any loop could be placed inside the other loop

e.g$_1$: **#include<stdio.h>**

```
main()
{
int i,j;
clrscr();
for(i=1;i<=5;i++)
{
for(j=1;j<=i;j++)
        printf("%d\t",j);
printf("\n");

}
        getch();
         }
```

The output is:

```
1
1   2
1   2   3
1   2   3   4
1   2   3   4   5
```

e.g$_2$:

```
#include<stdio.h>
main()
{
        int i,j;
```

```
                for(i=1;i<=5;i++)
                {
                        for(j=1;j<=i;j++)

        printf("%d\t",i);
        printf("\n");

                }
        }
```

The output is:

```
 1
 2   2
3   3   3
4   4   4   4
5   5   5   5   5
```

e.g3:

```
        #include<stdio.h>
        main()
        {
                int i,j;
                for(i=1;i<=5;i++)
                {
                        for(j=5;j>=i;j--)

        printf("%d\t",j);
        printf("\n");

                }
        }
```

The output is:

```
5   4   3   2   1
5   4   3   2
5   4   3
5   4
5
```

e.g4:

```
        #include<stdio.h>
        main()
        {
                int i,j;
                for(i=1;i<=5;i++)
```

```
        {
                for(j=5;j>=i;j--)

    printf("%d\t",i);
    printf("\n");

        }
    }
```

The output is:

```
1   1   1   1   1
2   2   2   2
3   3   3
4   4
 5
```

Exercises

Write an application to display

a)

```
 =   =   =   =   =
=   =   =   =
=   =   =
=   =
 =
```

b)

```
[*]
[*]  [*]
[*]  [*]  [*]
[*]  [*]  [*]  [*]
[*]  [*]  [*]  [*]  [*]
```

c) Write a program to display the table bellow:

```
1*1=1
2*2=4
3*3=9
4*4=16
5*5=25
6*6=36
7*7=49
8*8=64
9*9=81
    10*10=100
```

d)Write a program to generate Fibonacci series

HINT: Fibonacci series is the one in which a number is equal to the sum of the two previous numbers as follows 1 1 2 3 5 8 13 21 34 . . . . . .

While loop

The while provides a mechanism for repeating C statements whilst a condition is true.
While loop has one control expression and executes as long as that expression is true.

General structure (its format is):

```
while(expression)
{
statements;
}
```

Normally, whatever operations are carried out by the while loop can also be done by using for loop. Somewhere within the body of the while loop, a statement must alter the value of the condition to allow the loop to finish.

The for and while loop make a test of the conditions before the loop is executed.
Therefore, the body of the loop may never be executed at all if the conditions are not satisfied.

/* Sample program including while */

```
#include<stdio.h>
main()
{
int loop=0;
while(loop<=10)
{
printf("%d\n",loop);
++loop;
}
}
```

Sample program

```
0
1
2
..

10
```

The above program uses a while loop to repeat a statement.

```
printf("%d\n",loop);
        ++loop;
```

whilst the value of the variable loop is less than or equal to 10.

SIMPLE INTEREST CALCULATION USING WHILE LOOP

```
#include<stdio.h>
void main()
{
int p,n,count=1;
float rate,si;
clrscr();
while(count<=4)
{
printf("\n ENTER P N AND R\n");
scanf("%d%d%f",&p,&n,&rate);
si=(p*n*rate)/100,
printf("\nSIMPLE INTEREST IS =%f",si);
count++;
}
getch();
}
```

Exercises

e.g1:

Let's say that you would like to create a program that prints a Fahrenheit-to-Celsius conversion table.

```
#include <stdio.h>
int main()
{
    int a;
    a = 0;
    while (a <= 100)
    {
        printf("%4d degrees F = %4d degrees C\n",
            a, (a - 32) * 5 / 9);
        a = a + 10;
    }
    return 0;
}
```

If you run this program, if will produce a table of values starting at 0 degrees F and ending at 100 degrees F. The output will look like this:

```
  0 degrees F =  -17 degrees C
 10 degrees F =  -12 degrees C
 20 degrees F =   -6 degrees C
 30 degrees F =   -1 degrees C
 40 degrees F =    4 degrees C
 50 degrees F =   10 degrees C
 60 degrees F =   15 degrees C
 70 degrees F =   21 degrees C
```

```
 80 degrees F =     26 degrees C
 90 degrees F =     32 degrees C
100 degrees F =     37 degrees C
```

e.g2: Program that reads a set of number from keyboard and calculate their and average

```
#include<stdio.h>
main()
{
int i,n,a,sum=0;
float av;

printf("How many numbers:");
scanf("%d",&n);
i=0;
sum=0;
while(i<=n-1){
printf("Enter a number:");
scanf("%d",&a);
sum=sum+a;
++i;
av=(float)sum/n;
}
printf("The sum is %d\n",sum);
printf("average=%.2f",av);

}
```

e.g3: Using while for multiplication table from 1 to 10

```
#include<stdio.h>
main(){
int i,j;
i=1;
while(i<=10){
j=1;
while(j<=10){
printf("%d x %d = %d\n",i,j,i*j);
j++;
}
printf("\n\n");
i++;
}
}
```

do while

When developing programs, it sometimes become desirable to have the test made at the end of the loop rather than at the beginning.

Naturally, C language provides a special language construct to handle such a situation. This looping statement is known as the do statement.

The syntax is:

```
 do
{
statements;
}
while(expression)
{
statements;
}
```

Execution of the do statement proceeds as follows:
Program statement is executed first. Next, the expression inside the parenthesis is evaluated, if the result of evaluating is TRUE, the loop continues and the program statement is once again executed. As long as evaluation of expression continues to be true, program statement is repeatedly executed. When evaluation of the expression is FALSE, the loop is terminated and the next statement in the program is executed in the normal sequential manner.

The do statement is simply a transposition of the while loop statement with the looping conditions placed at the end of the loop rather than at the beginning.

N.B: Remember that, unlike the for and while loops, the do statement guarantees that the body of the loop will be executed at least once.

e.g$_1$: Program that displays number from 1 to 9 using Do while

```
#include<stdio.h>
main()
{
int i;
clrscr();
i=1;
do
{
printf("\n i is:%d",i);
i=i+1;
}
while(i<10);
getch();
}
```

e.g$_2$: program that displays number from 1 to 30 with steps=3 using Do while

```
#include<stdio.h>
main()
{
int x;
clrscr();
printf("number from 1 to 30\n");
x=1;
do
{
printf("%d\t",x);
x=x+3;
}
while(x<=30);

getch();
}
```

eg 3 SIMPLE INTEREST USING DO-WHILE LOOP

```
#include<stdio.h>
void main()
{
int p,n;
float rate,si;
char yes;
clrscr();
do
{
printf("\n ENTER P N AND R\n");
scanf("%d%d%f",&p,&n,&rate);
si=(p*n*rate)/100,
printf("\nSIMPLE INTEREST IS =%f",si);
printf("\n DO YOU WANT TO CONTINUE?SAY Y/N");
scanf("%s",&yes);
}
while(yes=='y'||yes=='Y');
getch();
}
```

eg 4 .CALCULATION OF ROOTSOF QUADRATIC EQUATION USING DO- WHILE LOOP

```
#include<stdio.h>
#include<math.h>
main()
```

```
{
 float a,b,c;
 char yes;
 clrscr();
 do
 {
 printf("ENTER THREE CONSTANTS FOR A QUADRATIC EQUATION\n");
 scanf("%f%f%f",&a,&b,&c);
 if(a==0)
 {
 printf("THIS IS NOT A QUADRATIC EQUATION\n");
 printf("THE VALUE OF X =%f",(-c/b));
 }
 else if(((b*b)-(4*a*c))==0)
 {
 printf("THE QUADRATIC EQUATION HAS TWO EQUAL ROOTS\n");
 printf("TWO EQUAL ROOTS ARE ROOT1=ROOT2=%f",(-b/(2*a)));
 }
 else if(((b*b)-(4*a*c))>0)
 {
 printf("THE QUADRATIC EQUATION HAS TWO DISTINCT ROOTS\n");
 printf("THE FIRST ROOT IS ROOT1=%f\n",(-b-sqrt(b*b-4*a*c))/(2*a));
 printf("THE SECOND ROOT IS ROOT2=%f\n",(-b+sqrt(b*b-4*a*c))/(2*a));
 }
 else
 {
 printf("THE EQUATION HAS COMPLEX ROOTS\n");
 printf("THE FIRST COMPLEX ROOT1=%f-j%f\n",-b/(2*a),sqrt(-(b*b-4*a*c))/(2*a));
 printf("THE SECOND COMPLEX ROOT2=%f+j%f\n",-b/(2*a),sqrt(-(b*b-4*a*c))/(2*a));
 }
 printf("\n DO YOU WANT TO CONTINUE?SAY Y/N");
 scanf("%s",&yes);
 }
while(yes=='y'||yes=='Y');
 getch();
}
```

Chapter 6 ARRAYS

-Arrays are data structures that hold multiple variables of the same data type, stored in a consecutive memory location in common heading.

-Array is a set of similar data (homogeneous data items) that shares the common name.

-The individual values in the array are called as elements.

-An array lets you declare and work with a collection of values of the same type.

For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

```
int a, b, c, d, e;
```

This is okay, but what if you needed a thousand integers?
An easier way is to declare an array of five integers:

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

The five separate integers inside this array are accessed by an **index**.
All arrays start at index zero and go to n-1 in C.
Thus, **int a[5];** contains five elements.
For example:

```
int a[5];

a[0] = 12;
a[1] = 9;
a[2] = 14;
a[3] = 5;
a[4] = 1;
```

One of the nice things about array indexing is that you can use a loop to manipulate the index. For example, the following code initializes all of the values in the array to 0:

```
int a[5];
int i;

for (i=0; i<5; i++)
    a[i] = 0;
```

Declaring arrays

Arrays may consist of any of the valid data types. Arrays are declared along with all other variables in the declaration section of the program.

Array declaration is defining the type of array, name of the array, number of subscripts (whether is one or multi-dimensional)

```
/*  Introducing array's  */
        #include <stdio.h>
        main()
        {
                int    numbers[100];
                float  averages[20];

                numbers[2] = 10;
                --numbers[2];
                printf("The 3rd element of array numbers is %d\n", numbers[2]);
        }
```

Sample Program Output

```
The 3rd element of array numbers is 9
```

The above program declares two arrays, assigns 10 to the value of the 3rd element of array numbers, decrements this value (--numbers[2]), and finally prints the value. The number of elements that each array is to have is included inside the square brackets.

Assigning initial values to arrays

The declaration is preceded by the word static.(initialization is preceded by the word static but this one is optional) The initial values are enclosed in braces

E.g:

```
#include <stdio.h>
main()
        {
        int x;
                static int   values[] = { 1,2,3,4,5,6,7,8,9 };
                static char  word[] = { 'H','e','l','l','o' };
                for( x = 0; x < 9; ++x )
                        printf("Values [%d] is %d\n", x, values[x]);
        }
```

Sample Program Output

```
Values[0] is 1
Values[1] is 2
....
Values[8] is 9
```

The previous program declares two arrays, values and word. Note that inside the square brackets there is no variable to indicate how big the array is to be. In this case, C initializes the array to the number of elements that appear within the initialize braces. So values consist of 9 elements (numbered 0 to 8) and the char array word has 5 elements.

The following code initializes the values in the array sequentially and then prints them out:

e.g$_1$:

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

```
main()
{
    int a[5];
    int i;

    for (i=0; i<5; i++)
        a[i] = i;
    for (i=0; i<5; i++)
        printf("a[%d] = %d\n", i, a[i]);
    }
```

e.g2:

```
#include<stdio.h>
main()
{
int a[7]={11,12,13,14,15,16,17};
int i;
clrscr();
printf("Contents of the array\n");
for(i=0;i<=6;++i)
printf("%d\t",a[i]);
getch();
}
```

e.g3: Program that reads 10 numbers from keyboard and find out the minimum and the maximum.

```
#include<stdio.h>
main()
{
int i,x[10],max,min;
clrscr();
for (i=0;i<10;i++)
{
printf("Enter number:");
scanf("%d",&x[i]);
}
        min=x[0];
                for(i=1;i<10;i++)
{
            if(x[i]<min)
        min=x[i];
}
          max=x[0];
                for(i=1;i<10;i++)
{
            if(max<x[i])
          max=x[i];
```

```
}
printf("Min=%d\n",min);
printf("Max=%d\n",max);
getch();
}
```

e.g4:

/* A program to read a set of numbers from keyboard and to find out the largest number of the given array where the numbers are stored in a random order */

```
#include<stdio.h>
main()
{
        int a[100];
        int i,n,large;
        clrscr();
        printf("How many numbers in the array?\n");
        scanf("%d",&n);
        printf("Enter the elements:\n");
          for(i=0;i<=n-1;++i)
                {
                scanf("%d",&a[i]);
                }
        printf("Contents of the array\n");
          for(i=0;i<=n-1;++i)
                {
                printf("%d\t",a[i]);
                }
                printf("\n");
        large=a[0];
         for(i=0;i<=n-1;++i)
                {
           if (large<a[i])
        large=a[i];
                }
        printf("Largest value in the array = %d\n",large);
getch();
}
```

N.B: -Array whose elements are specified by one subscript are called one dimensional array or single dimensional array.

-The maximum size of the array is 200 elements. If you avail more than the declared size then the compiler will treat only the first n elements as significant.

- The subscript used to declare an array is sometimes called a dimension and the declaration for the array is often referred to as dimensioning.

- The dimension used to declare an array must always be a positive integer constant, or an expression that can be evaluated to a constant when the program is compiled

Multi Dimensioned Arrays

Multi-dimensioned arrays have two or more index values, which specify the element in the array.

multi[i][j]

In the above example, the first index value i specifies a row index, whilst j specifies a column index.

Declaration and calculations

```
int         m1[10][10];
            static int m2[2][2] = { {0,1}, {2,3} };

            sum = m1[i][j] + m2[k][l];
```

NOTE the strange way that the initial values have been assigned to the two-dimensional array m2. Inside the braces are,

{ 0, 1 },
{ 2, 3 }

Remember that arrays are split up into row and columns. The first, is the row, the second is the column. Looking at the initial values assigned to m2, they are,

m2[0][0] = 0
m2[0][1] = 1
m2[1][0] = 2
m2[1][1] = 3

Now, consider the following array declaration:

int values[3][4] ={1,2,3,4,5,6,7,8,9,10,11,12};

The result of this initial assignment is as follows:

values[0][0]=1	values[0][1]=2	values[0][2]=3	values[0][3]=4
values[1][0]=5	values[1][1]=6	values[1][2]=7	values[1][3]=8
values[2][0]=9	values[2][1]=10	values[2][2]=11	values[2][3]=12

This, can be initialized by forming groups of initials values enclosed within braces

int values[3][4] ={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

While initializing a two dimensional array, it is necessary to mention the second (column) dimension, the first dimension (row) is optional.

Thus, the declarations given below are perfectly acceptable.

int arr[3][4] ={12,34,23,45,56,45};
int arr[ ][3] ={12,34,23,45,56,45};

However, the following declarations will never work

```
int arr[2][ ] ={12,34,23,45,56,45};
int arr[ ][ ] ={12,34,23,45,56,45};
```

e.g$_1$: Program that reads a 4*4 matrix numbers from keyboard and displays it.

```
#include<stdio.h>
main()
{
int a[4][4];
int i,j,s;
clrscr();
printf("Enter the 4*4 Matrix\n");
printf("_____\n");
for(i=1;i<=4;i++)
        for(j=1;j<=4;j++)
                scanf("%d",&a[i][j]);
                printf("The matrix entered is:\n");
                        for(i=1;i<=4;i++)
                                for(j=1;j<=4;j++)
                        printf("a[%d,%d]=%d\n",i,j,a[i][j]);
getch(); }
```

e.g$_2$: Program that prints the months depending of the numer entered

```
#include<stdio.h>
main()
{
char month[][30]={"january","february","march","april","may","june","july",
"august","september","october","november","december"};
int mon_num;
clrscr();
printf("Enter the month number:");
scanf("%d",&mon_num);
if((mon_num>=0)&&(mon_num<12))
{
printf("\nMonth corresponding to number %d is
%s",mon_num,month[mon_num-1]);
}
else
printf("\n Invalid Month" );
getch();
}
```

e.g$_3$: Program that adds two matrix

```
#include<stdio.h>
# define MAX_ROWS 10
# define MAX_COLS 10
main()
{
```

```
int tot_row,tot_col,row,col;
int a[MAX_ROWS][MAX_COLS], b[MAX_ROWS][MAX_COLS];
int c[MAX_ROWS][MAX_COLS];
clrscr();
printf("How many rows ?:");
scanf("%d",&tot_row);
printf("How many columns:");
scanf("%d",&tot_col);
/*Input tables */
printf("\n Input for the first Tables\n");
for(row=0;row<tot_row;row++)
{
printf("\n Enter data for row no.%d\n",row+1);
for(col=0;col<tot_col;col++)
{
scanf("%d",&a[row][col]);
printf("\n");
}
}
printf("\n Input for the second table \n");
for(row=0;row<tot_row;row++)
{
printf("\n Enter data for row no.%d\n",row+1);
for(col=0;col<tot_col;col++)
{
scanf("%d",&b[row][col]);
printf("\n");
}
}
for(row=0;row<tot_row;row++)
{
for(col=0;col<tot_col;col++)
{
c[row][col]=a[row][col]+b[row][col];
}
}
/* Output the result Table */
printf("\n Sum of the elements :\n\n");
for(row=0;row<tot_row;row++)
{
for(col=0;col<tot_col;col++)
{
printf("%d\t",c[row][col]);
}
printf("\n");
}
```

```
getch();
}
```

Character Arrays [Strings]

Consider the following program,

```
#include <stdio.h>
main()
{
        static char name1[] = {'H','e','l','l','o'};
        static char name2[] = "Hello";
        printf("%s\n", name1);
        printf("%s\n", name2);
}
```

Sample Program Output

```
Helloxghifghjkloqw30-=kl`'
Hello
```

The difference between the two arrays is that name2 has a null placed at the end of the string, ie, in name2[5], whilst name1 has not. This can often result in garbage characters being printed on the end. To insert a null at the end of the name1 array, the initialization can be changed to,

```
static char name1[] = {'H','e','l','l','o','\0'};
```

Chapter 7 FUNCTIONS

-A function is a self-contained block of statements that perform coherent task; it supports the concept of modular programming design techniques. Every C program can be thought as a collection of these functions.

-The complex problem may be decomposed into small or easily manageable parts and each module is called a function.

-The functions are very useful to read, write, debug, modify and they are easy to use in the main program.

-In C programming, the main () itself is a function, this means that the main function is invoking the other functions to perform the task.

The main body of a C program, identified by the keyword main, and enclosed in by the left and right parentheses is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

There are basically 2 types of function:

-Library functions: **e.g:** printf(),scanf(),…

-User defined functions: **e.g:** message(),…

Library functions: are functions already known by the machine that is built in the compiler.

Libraries are very important in C because the C language supports only the most basic features that it needs. The codes are often placed in **libraries** to make them easily reusable. We have seen the standard I/O, or **stdio**, library already: Standard libraries exist for standard I/O, math functions, string handling, and so on. You can use libraries in your own programs to split up your programs into modules. This makes them easier to understand, test, and debug, and also makes it possible to reuse code from other programs that you write.

Some Built in Functions for String Handling

string.h

The following macros are built into the file string.h

strcmp	compares two strings
strcpy	copies one string to another
strlen	finds length of a string
strlwr	converts a string to lowercase
strncat	appends n characters of string
strrev	reverses string
strupr	converts string to uppercase

e.g$_1$:

```
/*To convert a string to uppercase */
        #include <stdio.h>
        #include <string.h>
```

```
main()
{
        char name[80];      /* declare an array of characters 0-79 */
        printf("Enter in a name in lowercase\n");
        scanf( "%s", name );
        strupr( name );
        printf("The name is uppercase is %s", name );
}
```

Sample Program Output

```
Enter in a name in lowercase
samuel
The name in uppercase is SAMUEL
```

e.g2:

```
#include<stdio.h>
#include<string.h>
main()
{
        char name[25];
printf("Enter name in upper case:\n");
scanf("%s",&name);
strlwr(name);
printf("The name in uppercase is %s\n",name);
strrev(name);
printf("The name in reverse is %s\n",name);
printf("The length of name is:%d\n",strlen(name));
        }
```

Sample Program Output

```
Enter in a name in uppercase
SAMUEL
The name in uppercase is Samuel
The name in reverse is leumas
The length of name is:6
```

Some built in Functions for Character Handling

The following character handling functions are defined in ctype.h

tolower	converts character to lowercase
toupper	converts character to uppercase

/* To convert a string array to uppercase a character at a time using toupper() */

```
#include <stdio.h>
#include <ctype.h>
main()
{
        char name[80];
        int loop;
        printf("Enter in a name in lowercase\n");
        scanf( "%s", name );
        for( loop = 0; name[loop] != 0; loop++ )
                name[loop] = toupper( name[loop] );
        printf("The name is uppercase is %s", name );
}
```

Sample Program Output

```
Enter in a name in lowercase
samuel
The name in uppercase is SAMUEL
```

Some built in Functions for Math Functions

sqrt()	calculate the square root of a number
cos()	calculate the cosine
sin()	calculate the sine
tan()	calculate the tangent
abs()	calculate the absolute value
fabs()	calculate the absolute value of a floating point
ceil()	is used to round up to an integer
floor()	is used to round down to an integer
exp()	is used to compute a floating point approximation to the exponential function
log()	is used to compute a floating point approximation to the exponential function
log10()	is used to compute a floating point approximation to the natural logarithms function
pow()	is used to compute the power of a number.

e.g$_1$:

```
#include<stdio.h>
#include<math.h>
main()
{
float a,b,pi=22/7;
printf("Enter a number:\n");
scanf("%f",&a);
```

```
printf("The square root of %.2f is %.3f\n",a,sqrt(a));
printf("The power of %.2f is %.3f\n",a,pow(a,2));
printf("Please enter any number in radian:\n");
scanf("%f",&b);
if (b = = 11/7) /* this means (22/7)/2 approximate equal to 1.571428 */
{
printf("Error");
}
else
{
printf("The cosine of %.6f radian is :%.3f\n",b, cos(b));
printf("The sine of %.6f radian is :%.3f\n",b ,sin(b));
printf("The tangent of %.6f radian is :%.3f\n",b ,tan(b));
}
}
```

e.g$_2$:

```
/* A program to calculate the volume of a sphere */
#include<stdio.h>
#include<math.h>
#define pi 3.14
main()
{
float r,v;
printf("Enter the radius:\n");
scanf("%f",&r);
v=(float)4*pow(r,3)*pi/3;
printf("The volume is:%.2f\n",v);
}
```

User Defined Functions: they are those functions developed by the user.

In function we have:

- A program that calls or activates the function and
- The function itself

e.g$_1$:

```
#include<stdio.h>
main()
{
message();/* Calling of a function */
printf("This is a C function\n");
}
message()          /*  Function declaration*/
{
printf("\n Message function");
```

```
}
```
Output

```
Message function
This is a C function
```

Here, through main() we are calling the function message(). What do we mean when we say that main() calls the function message()?
We mean that the control passes to the function message(). The activity of main() is temporarily suspended; it falls asleep while the message() function makes up and goes to work. When the message runs out of statements to execute, the control returns to main(), which comes to life again and begins executing its code at the exact point it left off. Thus, main() becomes calling function, whereas message() becomes the called function.

Through the function message() did not have any arguments within the pair of parentheses, some functions may have them.
If the functions are present, before beginning with the statements in functions it is necessary to declare the types of arguments (parameters for receiving inputs) through type declaration statements.

The general form of a function:

```
function_name(arg1,arg2,….argn)
type arg1,arg2,…argn;
{
statements;
statements;
……..
}
```

N.B: A function definition has 2 parts:
-argument declaration
-body of the function

A function declaration has a name and a parentheses pair () containing zero or more parameters and a body. For each parameter, there should be a corresponding declaration that occurs before the body.

e.g2:

```
#include<stdio.h>
main()
{
printf(“Country\n”);
rwanda();
congo();
kenya();
```

```
}
rwanda()
{
printf("Rwanda\n");
}
congo()
{
printf("Congo\n");
}
kenya()
{
printf("Kenya\n");
}
```

Output

```
Country
Rwanda
Congo
Kenya
```

From this program a number of conclusion can be taken:

- Any C program contains at least one function
- If a program contains one function, it must be main()
- In a C program if there are more than one functions present, then one (and only one) of these functions must be main(), because program execution always begins with main().
- There is no limit on number of functions that might be present in a C Program
- Each function in a program is called in the sequence specified by the function calls in main().
- After each function has done its thing, control returns to main, when main() runs out of function calls, the program ends.
- The program execution always begins with main()

Summary

1. C program is a collection of one or more functions.
2. A function gets called when is followed by a semicolon
3. A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

   e.g:
   ```
   kist()
   {
   statement;
   statement;
   }
   ```

4.Any function can be called from any other function.

e.g:

```
#include<stdio.h>
main()
{
message();
}
message()
{
printf("\n C Programming\n");
}
```

5.A function can be called any number of times

e.g:

```
#include<stdio.h>
main()
{
message();
message();
}
message()
{
print("\n C Programming\n");
}
```

6.The order in which the functions are defined in a C program and the order in which they get called need not necessary to be same but for logical output it should be respected.

e.g :

```
#include<stdio.h>
main()
{
message1();
message2();
}
message2()
{
printf("\n C Programming\n");
}
message1()
{
printf("Is easy\n");
}
```

7.A function can call itself. Such process is called "**recursion**".

Recursive function

In C, it is possible for the functions to call themselves. A function is called "**recursive**" if a statement within the body of a function calls the same function (sometimes called circular definition).

Recursion is thus the process of defining something in terms of itself.

e.g: /*Recursive function to calculate the factorial value */

```
#include<stdio.h>
main()
{
int a,fact;
printf("Enter any number\n");
scanf("%d", &a);
fact=rec(a);
printf("Facrorial number of   %d is: %d",a,fact);
}
rec()                    // or rec(int x)
int x;
{
int f;
if(x= =1)
return(1);
else
f=x*rec(x-1);
return(f);
}
```

8.A function can be called from other function. Thus the following program code would be wrong, since argentina() is being defined inside another function, main()

e.g :

```
main()
{
print("\n I am in main");
argentina()
{
printf("\n I am argentina\n");
}
}
```

Advantages of using functions

1. Writing functions avoids rewriting the same code over and over.
2. Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of the program can be divided into separate activities and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

Passing values between functions

Consider the following program

```
#include<stdio.h>
 main()
{
int a,b,sum;
printf("Enter 2 numbers:\n");
scanf("%d %d",&a,&b);
sum= calsum(a,b);
printf("Sum=%d\n",sum);
}
calsum(x,y)              // or calsum(int x, int y)
{
int d;
d=x+y;
printf("The following is:\n");
return(d);
}
```

- In this program, in main() we receive the values of a,b through the keyboard and then output the sum of a and b. However the calculation of sum is done in different function called calsum(). If sum is to be calculated in calsum() and values of a and b are received in main(), then we must pass on these values to calsum(), and once calsum() calculates the sum we must return it from calsum() back to main().
- Values a,b are passed to function calsum(), by making a call to the function calsum() and mentioning a, b in the parentheses: sum=calsum(a,b);
- In the calsum() function these values get called in two variables x and y

  ```
  calsum(x,y)
  int x,y;
  ```
- The variables a and b are called "**actual arguments**", whereas the variables x y are called "**formal arguments**". Any number of arguments can be passed to a function being called. The order, type and number of the actual and formal arguments must always be same.
- Instead of using different variable names x,y we could have used the same variable names a and b. But the compiler would still treat them as different variables since they are in different function variables since they are in different functions.

- The 2 methods of declaring the formal arguments are the same:

```
calsum()
int x,y;
        and
calsum(int x,inty)→ commonly used
```

- In the above program if we want to return the sum of x, y, it is necessary to use the **return** statement.
- The return statement serves 2 purposes:
    1. On executing the return statement it immediately transfers the control back to the calling program.
    2. It returns the value present in the parentheses after return to the calling program. In the above program the value of sum of 2 numbers is being returned.

- If we want that a called function should not return any value, in that case, we must mention so by using the keyword **void**.

```
void message()
{
printf("C programming is easy\n");
}
```

- A function can return only one value at time. The following statement is invalid: return(a,b);

Function declaration and Prototypes

-Any C function by default returns an int value. More specifically whenever a call is made to a function, the computer assumes that this function would return a value of the type int.
If we desire that a function should return a value other than in int, then it is necessary to explicitly mention so in the calling function as well as in the called function.

-Function prototypes are listed at the beginning of the source file. Often, they might be placed in a users header file.

-Any parameter not declared is taken to be an int by default.

-It is a good programming practice to declare all parameters

```
#include<stdio.h>
main()
{
float a,b;
printf("Enter any number:\n");
scanf("%f",&a);
b=square(a);
printf("Square of %.2f=%.2f\n",a,b);
```

```
}
square(x)
float x;
{
float y;
y=x*x;
return(y);
}
```

If you enter 1.2, output will be 1.00

Look at the following program.

```
#include<stdio.h>
main()
{
float square();
float a,b;
printf("Enter any number:\n");
scanf("%f",&a);
b=square(a);
printf("Square of %.2f=%.2f\n",a,b);
}
square(x)
float x;
{
float y;
y=x*x;
return(y);
}
```

If you enter 1.5, output will be 2.25

This is because the function square() has been declared as float in main().
The statements float square() means that it is a function which will return a float value.
In case we want that a called function should not return any value, we use the keyword **void**.

Storage classes (Scope of variables)

In C, all the variables have data types and storage classes. The scope of variable refers to how widely it is known among a set of functions in a program. Every identifier also has a storage class that provides its visibility, lifetime and location. There are 4 different storage classes in C. They are:

- Automatic storage class
- Register storage class
- Static storage class
- Extern storage class

Automatic storage class

Variables defined inside a function are local (internal) to the function they are declared, and called automatic variables. They are more often referred to as automatic, after the fact that their memory space is automatically allocated as the function is entered and released.
In other words, automatic variables are given only temporary memory space. They are only accessed by the function in which it is defined. They have no meaning outside the function in which they are declared.
The portion of the program where a variable can be used is called the scope of that variable.

The C compiler treats any variable declared inside a function as an automatic variable, it is not necessary to specify the keyword **auto** along with the variable declaration.

e.g1:

```
#include<stdio.h>
main()
{
        auto int x=5;
        clrscr();
          {
            auto int x=4;
              {
                auto int x=3;
                 printf("%d\t",x);
              }
               printf("%d\t",x);
        }
printf("%d\t",x);
}
```

The output will be 3 4 5

e.g**2:**A program of displaying 1 to 10 with 100 using the automatic variable

```
#include<stdio.h>
main()
{
int i;
for(i=1;i<=10;i++)
printf("%d %d\n",i,f(i));
}
f(x)
int x;
{
int s=100;                    /*automatic variable */
return (s+=x);                /* x=x+s*/
}
```

Output

1	101	(i.e 1+100)
2	102	(i.e 2+100)
3	103	(i.e 3+100)
4	104	(i.e 4+100)
5	105	(i.e 5+100)
6	106	(i.e 6+100)
7	107	(i.e 7+100)
8	108	(i.e 8+100)
9	109	(i.e 9+100)
10	110	(i.e 10+100)

Register storage class

The register storage class is similar to the previous one since the variables defined inside a function are local. It can be accessed by the function in which it is defined .Its values cannot be accessed by any other function. When defining a register variable inside a function it is more precise in C to use the keyword **register** before the definition of the variable.
One can request the computer to keep a limited number of variables in their registers for fast processing. The machine register sometimes called accumulators would increase the speed.

e.g3:

```
#include<stdio.h>
main()
{
register int x;
for(x=1;x<=10;x++)
printf("\n%d",x);
}
```

We cannot use the register storage class for all types of variables
i.e

```
register double x;
register float y;
```

The above declaration are wrong because the CPU register in the micro computers are usually 16 bit register and therefore cannot hold a float or a double value, which require 52 and 64 bytes respectively for storing a value. If the above declaration are used you won't get error messages, the compiler would treat the variables as automatic variables.

Static storage class

The static variables are stored in the memory. When the value of a static variable is not initialized it takes a value of zero.
The word static in general refers to anything that is inert to change. When defining static variable inside a function is more precise in C use the keyword **static** before the definition of the variable.

e.g4:

```
 #include<stdio.h>
main()
{
sum();
sum();
sum();
}
sum()
{
static int x=1;
printf("%d\t",x);
x=x+1;
}
```

Output
1 2 3

e.g5: :A program of displaying 1 to 10 with 100 using the static variable

```
#include<stdio.h>
main()
{
int i;
clrscr();
for(i=1;i<=10;i++)
printf("%d %d\n",i,f(i));
```

```
}
f(x)
int x;
{
static int s=100;
return s+=x;
}
```

Output

1	101	(i.e 1+100)
2	103	(i.e 2+101)
3	106	(i.e 3+103)
4	110	(i.e 4+106)
5	115	(i.e 5+110)
6	121	(i.e 6+115)
7	128	(i.e 7+121)
8	136	(i.e 8+128)
9	145	(i.e 9+136)
10	155	(i.e 10+145)

Since s has permanent memory space it keeps the same value in the period of time between leaving function f and again entering it later. In contrast to automatic variables, static variables are initialized only once. So here s has the value 100 only.

Extern storage class

Variables that are both alive and active throughout the entire program are called as external variables. They are also called global variables; they can be accessed by any function in a program. Global variables do not belong to any particular function.

When defining an extern variable or global variable in a program, it is more precise in C to use the keyword **extern** before the definition of the variable.

e.g$_6$:

```
#include<stdio.h>
int x;
main()
{
x=1;
f();

}
f()
{
printf("Value = %d",x);
}
```

output

Value=1

In the above program x has been declared as the global variable, in other words, external variable.
It is declared out of main function. One has to declare the external variables only once but these external variables can be used both main and function f() without declaring the data type again.

Chapter 8 STRUCTURES

A structure is a collection of variables, possibly of different types; grouped together under a single name for convenient handling.

Structures help to organize complicated data, particularly in large programs, because they permit a group related variables to be treated as a unit instead of separate entities.

The individual structure elements are referred to as members (field). Each member of a structure variable is specified by a variable name with a period and the member name. The period is a structure member operator, which we shall hereafter simplify as the period operator.

Let's create a new data structure suitable for storing the date. The elements or fields that make up the structure use the four basic data types. As the compiler cannot know the storage requirements for a structure, a definition for the structure is first required. This allows the compiler to determine the storage allocation needed, and also identifies the various sub-fields of the structure.

```
struct   date {
                int   month;
                int   day;
                int   year;
          };
```

This declares a new data type called date. This date structure consists of three basic data elements, all of type integer. This is a definition to the compiler**.** It does not create any storage space and cannot be used as a variable. In essence, it's a new data type keyword, like int and char, and can now be used to create variables. Other data structures may be defined as consisting of the same composition as the date structure,

```
struct  date   todays_date;
```

defines a variable called todays_date to be of the same data type as that of the newly defined data type struct date.

Assigning values to structure elements

To assign todays_date to the individual elements of the structure todays_date, the statement

```
todays_date.day = 31;
todays_date.month = 8;
todays_date.year = 2006;
```

is used. Note the use of the . element to reference the individual elements within todays_date.

e.g$_1$:

```
/* Program to illustrate a structure */
#include<stdio.h>
struct date                /* Global definition of type date*/
        {
int day;
int month;
int year;
        };
```

```
main()
        {
struct date today;
                today.day=31;
                today.month=8;
                today.year=2006;
clrscr();
printf("Today's date is: %d/%d/%d\n",today.day, today.month, today.year);
getch();
        }
```

e.g2:

```
#include<stdio.h>
struct date{
int day,month,year;
};
main()
{
static struct date dates[5];
int i;
//clrscr();
for(i=0;i<5;i++){
printf("Please enter the date (dd/mm/yy)\n");
scanf("%d/%d/%d",&dates[i].day,&dates[i].month,&dates[i].year);
}
//getch();
}
```

e.g3:

```
#include<stdio.h>
void main()
{
struct employee
 {
char fname[20],lname[20];
int age,id;
float weight,height;
 };

 struct employee emp;
printf("Enter employee ID:\n");
scanf("%d",&emp.id);
printf("Enter employee First Name and Last Name\n");
scanf("%s %s",&emp.fname,&emp.lname);
printf("Enter employee age:\n");
scanf("%d",&emp.age);
printf("Enter the Employee weight and height:\n");
```

```
scanf("%0.2f %0.2f",&emp.weight,&emp.height);
}
```

e.g$_4$: #include<stdio.h>

```
struct book
{
char name[25];
float price;
int page;
};
main()
{
struct book   b1, b2,b3;
//clrscr();
printf("Enter the name, price(in dollars) and the number of pages:\n");
scanf("%s %f %d",&b1.name,&b1.price,&b1.page);
scanf("%s %f %d",&b2.name,&b2.price,&b2.page);
scanf("%s %f %d",&b3.name,&b3.price,&b3.page);
printf("Name=%s\t, Price=%.2f\t,   Page= %d\n",b1.name,b1.price,b1.page);
printf("Name=%s\t, Price=%.2f\t,   Page=%d\n",b2.name,b2.price,b2.page);
printf("Name=%s\t, Price=%.2f\t,   Page=%d\n",b3.name,b3.price,b3.page);
//getch();
}
```

Output will be:

Enter the name, price and the number of pages:

Name=Tintin, Price=12.50, Page=62
Name=Kouakou, Price=3.50, Page=48
Name=C_Programming, Price=32.75, Page=148

N.B: - Normally, a structure is a heterogeneous data type whereas the array is homogenous data types

- The general syntax of a structure is as follows:

```
storage class struct user defined_name
{
data type member_name 1;
data type member_name 2;
data type member_name3;
…
…
data type member-name n;
}
variable1, variable2, variable3,…, variable n;
```

The structure definition is specified by keyword struct. This followed by a user-defined name surrounded by braces, which describes the members of the structures.

The storage class (static) is an optional. The keyword struct and the braces are required. The user-defined name is usually used, but there situations in which it is not required.

Chapter 9 POINTERS

A "normal variable" is a location in memory that can hold a value. For example, when you declare a variable **i** as an integer, four bytes of memory are set aside for it. In your program, you refer to that location in memory by the name **i**. At the machine level that location has a memory address. You, the programmer, know the four bytes at that address as **i**, and the four bytes can hold one integer value.

A pointer is different. A pointer is a variable that **points** to another variable. This means that a pointer holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. A pointer "points to" that other variable by holding a copy of its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That address points to a value. There is the pointer and the value pointed to.

The following example code shows a typical pointer:

e.g$_1$:

```
#include <stdio.h>
main()
{
int i,j;
int *p;                 /* a pointer to an integer */
p = &i;
 *p=5;
 j=i;
printf("%d %d %d\n", i, j, *p);
 return 0;
 }
```

The first declaration in this program declares two normal integer variables named **i** and **j**. The line **int *p** declares a pointer named **p**. This line asks the compiler to declare a variable **p** that is a **pointer** to an integer. The ***** indicates that a pointer is being declared rather than a normal variable. You can create a pointer to anything: a float, a structure, a char, and so on. Just use a ***** to indicate that you want a pointer rather than a normal variable.

The line **p = &i;** in C, **&** is called the **address operator**. The expression **&i** means, "The memory address of the variable **i**." Thus, the expression **p = &i;** means, "Assign to **p** the address of **i**." Once you execute this statement, **p** "points to" **i**. Before you do so, **p** contains a random, unknown address.

One good way to visualize what is happening is to draw a picture. After **i**, **j** and **p** are declared, they look like this:

In this drawing the three variables **i**, **j** and **p** have been declared, but none of the three has been

initialized. The two integer variables are therefore drawn as boxes containing question marks -- they could contain any value at this point in the program's execution. The pointer is drawn as a circle to distinguish it from a normal variable that holds a value, and the random arrows indicate that it can be pointing anywhere at this moment.

After the line **p = &I;**, **p** is initialized and it points to **i**, like this:

Once **p** points to **i**, the memory location **i** has two names. It is still known as **i**, but now it is known as ***p** as well. This is how C talks about the two parts of a pointer variable: **p** is the location holding the address, while ***p** is the location pointed to by that address. Therefore ***p=5** means that the location pointed to by **p** should be set to 5, like this:

Because the location ***p** is also **i**, **i** also takes on the value 5. Consequently, **j=i;** sets **j** to 5, and the **printf** statement produces **5 5 5**.

The main feature of a pointer is its two-part nature. The pointer itself holds an address. The pointer also points to a value of a specific type - the value at the address the point holds. The pointer itself, in this case, is **p**. The value pointed to is ***p**.

e.g2:

```
#include<stdio.h>
main()
{
int a=3;
printf("address of a= %d\n", &I);
printf("Value of a=%d\n", a);
printf("Value of a=%d\n", *(&a));
}
```

To define pointers requires 3 things:

1. Data type of the pointer
2. Pointer declaration
3. Name of the pointer variable

int *ptr
int * is a pointer data type
ptr is the name of the pointer

Advantages of pointers

- provides functions which can hold modify their calling arguments
- supports dynamic allocation routines
- improves the efficiency of certain routines

CHAP 10. File Handling

- When you need text I/O in a C program, and you need only one source for input information and one sink for output information, you can rely on **stdin** (standard in) and **stdout** (standard out).
- We distinguish two accessible memory types in computer.
    - Primary memory (RAM)
    - Secondary memory (Hard disk)
- We can store data in the disk so that we can retrieve it whenever and wherever we wish.
- Data in the disk is stored in units called files. We therefore say that files are the structures by which data is stored in disks (input) and is retrieved from the same (output).
- C provides input and output functions that can handle:
    - Single character
    - Entire line
    - Formatted text

There are six different I/O functions in <stdio.h> that you can use with stdin and stdout:

Function	**Read**	**Description**	**Write**	**Description**
Character (I/O)	**getc**, **getchar**	reads a character from stdin.	**Putc,putchar**	prints a character to stdout
String (I/O)	**gets**	reads a string from stdin.	**puts**	prints a string to stdout
Formatted (I/O)	**scanf**	reads formatted input from stdin	**printf**	prints formatted output to stdout

Standard in (stdin)→ keyboard

Standard out (stdout) → Screen

File Access Modes

In addition to reading characters from standard input and output, data can be read from a data file.

Before the data in file can be accessed, the file must be opened in the proper mode.

Mode	**Description**
"w"	Means write to the file, searches for it. if the file name exists already on the storage device, then it is deleted a new file will be created.

“r”	Searches file. And reads from it. The file must be already created. File is already for reading only. If the file does not exist it returns NULL.
“a”	Means append a file. Searches file. If it exists, loads it into memory and new data are added to the end of file.

File input/Output

Sometimes, you need to use a text file directly. For example, you might need to open a specific file and read from or write to it. You might want to manage several streams of input or output or create a program like a text editor that can save and recall data or configuration files on command. In that case, use the text file functions in stdio:

- **fopen** - opens a text file
- **fclose** - closes a text file
- **feof** - detects end-of-file marker in a file
- **fprintf** - prints formatted output to a file
- **fscanf** - reads formatted input from a file
- **fputs** - prints a string to a file
- **fgets** - reads a string from a file
- **fputc** - prints a character to a file
- **fgetc** - reads a character from a file

You use **fopen** to open a file. It opens a file for a specified mode (the three most common are r, w, and a, for read, write, and append). It then returns a file pointer that you use to access the file. For example, suppose you want to open a file and write the numbers 1 to 10 in it. You could use the following code:

```
#include <stdio.h>
#define MAX 10

int main()
{
    FILE *f;
    int x;
    f=fopen("out","w");
    if (!f)
        return 1;
    for(x=1; x<=MAX; x++)
        fprintf(f,"%d\n",x);
    fclose(f);
    return 0;
}
```

The **fopen** statement here opens a file named **out** with the w mode. This is a destructive write

mode, which means that if **out** does not exist it is created, but if it does exist it is destroyed and a new file is created in its place. The fopen command returns a pointer to the file, which is stored in the variable f. This variable is used to refer to the file. If the file cannot be opened for some reason, f will contain **NULL**.

Main Function Return Values

This program is the first program in this series that returns an error value from the main program. If the **fopen** command fails, f will contain a NULL value (a zero). We test for that error with the if statement. The if statement looks at the True/False value of the variable **f**. Remember that in C, 0 is False and anything else is true. So if there were an error opening the file, f would contain zero, which is False. The **!** is the **NOT** operator. It inverts a Boolean value. So the if statement could have been written like this:
if (f == 0)

That is equivalent. However, **if (!f)** is more common.

If there is a file error, we return a 1 from the main function. In UNIX, you can actually test for this value on the command line. See the shell documentation for details.

The fprintf statement should look very familiar: It is just like printf but uses the file pointer as its first parameter. The fclose statement closes the file when you are done.

To read a file, open it with r mode. In general, it is not a good idea to use **fscanf** for reading: Unless the file is perfectly formatted, fscanf will not handle it correctly. Instead, use **fgets** to read in each line and then parse out the pieces you need.

The following code demonstrates the process of reading a file and dumping its contents to the screen:

```
#include <stdio.h>
int main()
{
    FILE *f;
    char s[1000];

    f=fopen("infile","r");
    if (!f)
        return 1;
    while (fgets(s,1000,f)!=NULL)
        printf("%s",s);
    fclose(f);
    return 0;
}
```

The fgets statement returns a NULL value at the end-of-file marker. It reads a line (up to 1,000 characters in this case) and then prints it to stdout. Notice that the printf statement does not include \n in the format string, because fgets adds \n to the end of each line it reads. Thus, you can tell if a line is not complete in the event that it overflows the maximum line length specified in the second parameter to fgets.