Table of Content

1 - Introduction

1.1 Introduction to Programming

Software refers to a program or set of instructions that instructs a computer to perform some task. Software can be divided into two major categories called *system software* and *application software*. Systems software includes operating systems and various device drivers. Application software are used to perform real-world tasks and solve specific problems.

A *program* is simply *a* set of *instructions* that tells a computer how to perform a particular task. Programs are developed using *programming languages*. *Computer programming* is the art of developing computer programs. Programming is rather like a recipe; a set of instructions that tells a cook how to make a particular dish. It describes the ingredients (the *data*) and the sequence of steps (the *process*) on how to mix those ingredients.

A programming language provides a set of rules to develop a program. A person who writes a program using a programming language is called a *programmer*. His/her job is to convert a solution to a problem (i.e. algorithm) into set of instructions understood by a computer. The programmer should also test the program to see whether it is working properly and corrective actions should be taken if not.

1.2 Program Development

Developing a program involves a set of steps:

1. Define the problem
2. Outline the solution
3. Develop an algorithm[1]
4. Test the algorithm for correctness
5. Code the algorithm using a suitable programming language
6. Compile and correction of compile errors
7. Run the program on the computer
8. Test, document and maintain the program

Most of these steps are common to any problem solving task. Program development (software development) may take several hours, days, weeks, months or years. After development, customers will make use of the system. While in use, the system needs to be maintained. The maintenance phase will continue for several months, years or even several decades. Therefore software development is not a onetime task; it is a lifecycle where some of the above steps are reformed again and again. The steps are discussed in the following.

1.2.1 Define the Problem

First of all the problem should be clearly defined. The problem can be divided into three components:

- Inputs – what do you have?
- Outputs – what do you want to have?
- Processing – how do you go from inputs to outputs?

Programmers should clearly understand "what are the inputs to the program", "what is expected as output(s)" and "how to process inputs to generate necessary outputs". Consider an example where a computer program is to be written to calculate and display the circumference and area of a circle when the radious .

- Inputs – the radius (r)
- Outputs – circumference (c) and area (a)
- Processing
  - $c = 2pr$, $a = pr^2$

[1] An algorithm is a sequence actions that is used to solve a problem.

1.2.2 Outline the Solution

The programmer should define:

- the major steps required to solve the problem
- any subtasks
- the major variables and data structures
- the major control structures (e.g. sequence, selection, repetition loops)[2] in the algorithm
- the underlined logic

Consider the above mentioned example. In order to calculate the circumference:

- Variables – radius (r), circumference (c)
- Calculation – $c = 2pr$

In order to calculate the area:

- Variables – radius (r), area (a)
- Calculation – $a = pr^2$

1.2.3 Develop the Algorithm

The next step is to develop an *algorithm* that will produce the desired result(s). An algorithm is a segment of precise steps that describes exactly the tasks to be performed, and the order in which they are to be carried out to solve a problem. *Pseudocode* (a structured form of the English language) can be used to express an algorithm. A suitable algorithm for our example would be:

```
Start
Input r
Calculate circumference
      c = 2 * PI* r
Calculate area
      a = PI* r^2
Output c & a
End
```

In here **PI** is a constant that represents the value of **p.**

1.2.4 Test the Algorithm for Correctness

The programmer must make sure that the algorithm is correct. The objective is to identify major logic errors early, so that they may be easily corrected. Test data should be applied to each step, to check whether the algorithm actually does what it is supposed to.

Our simple example can be quite easily check by submitting some values for radius (r) and walking through the algorithm to see whether the resulting output is correct for each input.

1.2.5 Code the Algorithm

After all the design considerations have been met and when the algorithm is finalised code it using a suitable programming language.

1.2.6 Compile

The next step is to compile (section 1.3) the program. While compiling *syntax errors* can be identified. When the written program does not adhere to the programming language rules those are called syntax errors. These errors occur mostly due to miss typed characters, symbols, missing punctuations, etc. If there are no syntax errors the program gets compiled and it produces an executable program.

1.2.7 Run the Program

Executable program generated after compiling can then be executed. While the program is running *runtime errors* and sometimes *logic errors* can be identified. Runtime errors occur while executing the program and those are mostly due to incorrect inputs.

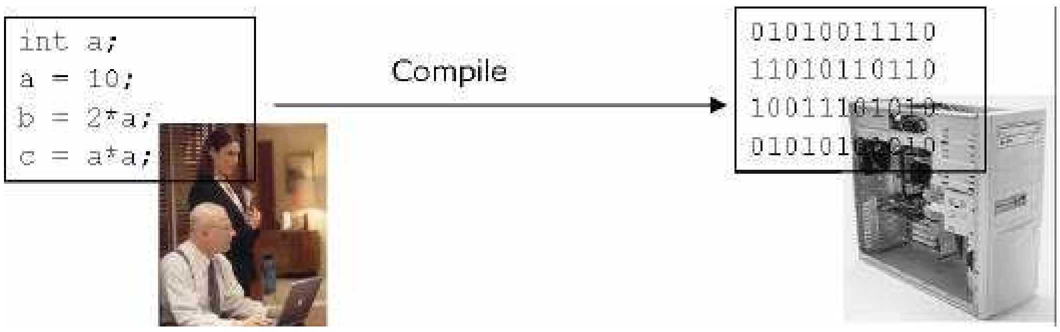
[2] Will be introduced later.

1.2.8 Test, Document and Maintain the Program

Test the running program using test data to make sure program is producing correct output(s). During this phase logic errors can be found. Logic errors occur due to incorrect algorithms (although you provide correct inputs you do not get the correct outputs). All the steps involved in developing the program algorithm and code should be documented for future reference. Programmers should also maintain and update the program according to new or changing requirements.

1.3 Running a Program

After a program is developed using a programming language, it should be executed by the computer. Programmers write programmes in human readable languages called *high-level languages*. However, the computer understands only 1's and 0's (referred as the *machine language*[3] or *machine code*). Therefore we need to convert the human readable programs into a machine language before executing. This is conversion is achieved by a special set of programs called *compilers* (or *interpreters*). These programs convert the high-level language program, into machine language programs.



```
int a;
a = 10;
b = 2*a;
c = a*a;
```

Figure 1.1 – Converting a human readable program into machine language

Compilers translate a source program (human-readable) into an executable (machine-readable) program (Figure 1.2). Executable machine code produced by a compiler can be saved in a file (referred as the *executable file*) and used whenever necessary. Since the source program is already compiled, no compilation is needed again unless the source program is modified. The saving on the compilation time is an advantage that is gain from compiling and therefore these programs run much faster. Programs written in programming languages such as FORTRAN, COBOL, C, C++ and Pascal must be compiled before executing.

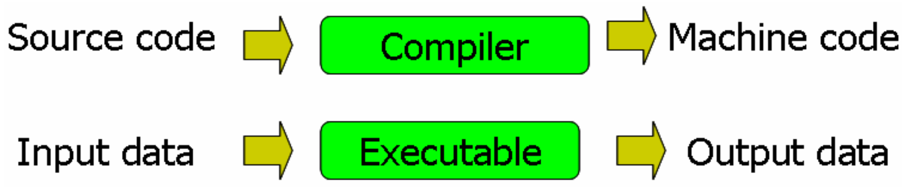


Figure 1.2 – Compiling and executing a program

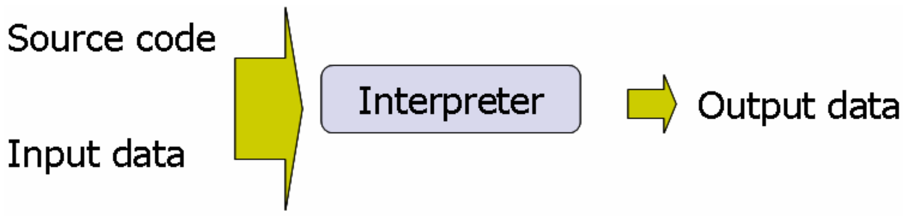


Figure 1.3 – Interpretation of a program

As in Figure 1.2 when the executable file is available it can be executed by providing necessary input data so that it produces the desired outputs.

Interpreters convert a source program and execute it at the same time (Figure 1.3). Each time the program runs, the interpreter converts high-level language instructions and input data to a machine readable format and executes the program. This process can be slower than the process which

[3] Machine language is the native language of the computer (i.e. what is understood by the hardware).

compiles the source program before execution. The program need to be converted as well as executed at the same time. Programs written in languages like BASIC, Perl, Smalltalk, Lisp and VB Script are interpreted.

1.4 Programming Languages

Programming languages were invented to make programming easier. They became popular because they are much easier to handle than machine language. Programming languages are designed to be both high-level and general purpose. A language is high-level if it is independent of the underlying hardware of a computer. It is general purpose if it can be applied to a wide range of situations. There are more than two thousand programming languages and some of these languages are general purpose while others are suitable for specific classes of applications. Languages such as C, C++, Java, C# and Visual Basic can be used to develop a variety of applications. On the other hand FORTRAN was initially developed for numerical computing, Lisp for artificial intelligence, Simula for simulation and Prolog for natural language processing (yet, these languages are general enough to be used for a wide range of problems.

As microprocessors, programming languages also can be grouped into several generations and currently we are in the fourth generation. In the early days, computers were programmed using machine language instructions that the hardware understood directly. Programs written using machine language belongs to the *first generation* of programming languages. These programs were hardly human readable, therefore understanding and modifying them was a difficult task.

Later programs were written in a human readable version of machine code called *Assembly language*. Assembly language belongs to the *second generation* of programming languages. Each Assembly language instruction directly maps into a machine language instruction (there is a 1-to-1 mapping). Assembly language programs were automatically translated into machine language by a program called an *assembler*. Writing and understanding Assembly language programs were easier than machine language programs. However even the Assembly language programs tend to be lengthier and tedious to write. Programs developed in Assembly runs only on a specific type of computer. Further, programmers were required to have a sound knowledge about *computer organization*[4].

With the introduction of *third generation* (also referred as 3GL) high-level languages were introduced. These languages allowed programmers to ignore the details of the hardware. The programs written using those languages were portable[5] to more than one type of hardware. A compiler or an interpreter was used to translate the high-level code to machine code. Languages such as FORTRAN, COBOL, C and Pascal belong to the third generation.

All the modern languages such as Visual Basic, VB Script, Java, C# and MatLab belong to the *fourth generation* (4GL). Programs written in these languages were more readable and understandable than the 3GL. They are much closer to natural languages. Source code of the programs written in these languages is much smaller than other generation of languages (i.e. a single high level language instruction maps into multiple machine language instructions). However, programs developed in 4GL generally do not utilize resources optimally. They consume large amount of processing power and memory and they are generally slower than the programs developed using languages belonging to other generations. Most of these 4GL support development of Graphical User Interfaces (GUIs) and responding to events such as movement of the mouse, clicking of mouse or pressing a key on the keyboard.

Some people think that *fifth generation* languages are likely to be close to natural languages. Such languages are one of the major research areas in the filed of Artificial Intelligence.

[4] Computer organization describes; the internal organization of a computer, the instruction set, instruction format, how to make use of registers, allocate and de allocate memory, talking to various I/O devices, etc.
[5] Running the same program on machines with different hardware configurations.

1.5 Overview of C

"C' seems a strange name for a programming language but it is one of the most widely used languages in the world. C was introduced by Dennis Ritchie in 1972 at Bell Laboratories as a successor of the language called B (Basic Combined Programming Language – BCPL). Since C was developed along with the UNIX operating system it is strongly associated with UNIX. UNIX was developed at Bell Laboratories and it was written almost entirely in C.

For many years, C was used mainly in academic environments. However with the release of C compilers for commercial use and increasing popularity of UNIX, it began to gain wide-spread interest among computer professionals. Various languages such as C++, Visual C++, Java and C# have branched away from C by adding object-orientation and GUI features. Today C compilers are available for a number of operating systems including all flavours of UNIX, Linux, MS-DOS, Windows and Apple Mac.

C is a robust language whose rich set of built-in functions and operations can be used to write any complex program. C is well suited to write both commercial applications and system software since it incorporates features of high-level languages and Assembly language. Programs written in C are efficient and fast. Most C programs are fairly *portable*; that is with little or no modification and compiling, C programs can be executed on different operating systems.

The syntax and coding style of C is simple and well structured. Due to this reason most of the modern languages such as C++, Java and C# inherit C coding style. Therefore it is one of the best languages to learn the art of programming. C is also suitable for many complex engineering applications.

1.6 Steps in Developing a Program in C

A programmer uses a text editor to create and modify files containing the C source code. A file containing source code is called a *source file* (C source files are given the extension `.c`). After a C source file has been created, the programmer must invoke the C compiler before the program can be executed. If the compiler finds no errors in the source code it produces a file containing the machine code (this file referred as the executable file).

The compilation of a C program is infact a three stages process; preprocessing, compiling and linking.

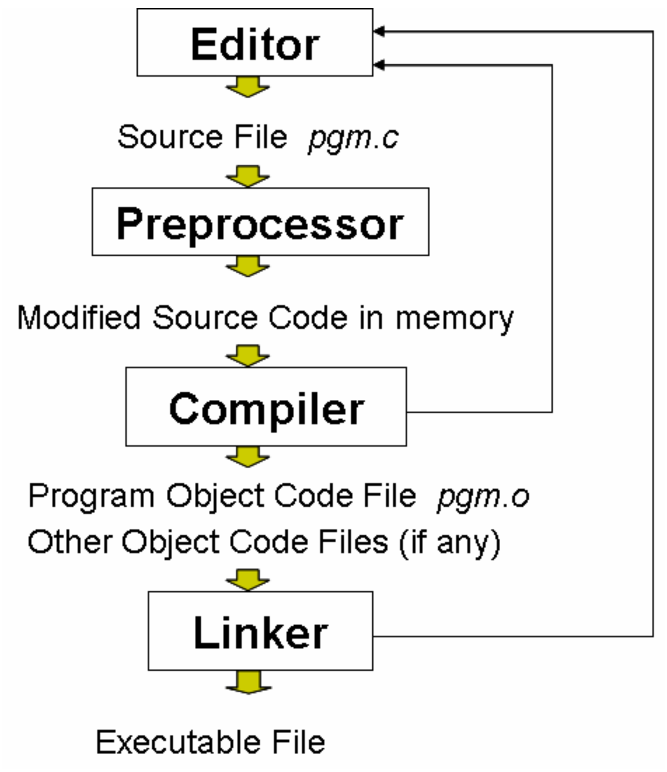

Editor
Source File *pgm.c*
Preprocessor
Modified Source Code in memory
Compiler
Program Object Code File *pgm.o*
Other Object Code Files (if any)
Linker
Executable File

Figure 1.4 – Compilation process

Preprocessing is performed by a program called the *preprocessor*. It modifies the source code (in memory) according to *preprocessor directives* (example: **#define**) embedded in the source code. It also strips comments and unnecessary white spaces from the source code. Preprocessor does not modify the source code stored on disk, every thing is done on the copy loaded into the memory.

Compilation really happens then on the code produced by the preprocessor. The compiler translates the preprocessor-modified source code into *object code* (machine code). While doing so it may encounter syntax errors. If errors are found it will be immediately notified to the programmer and compiling will discontinue. If the compiler finds any non-standard codes or conditions which are suspicious but legitimate it will notify to the programmer as warnings and it continues to compile. A well written program should not have any compilation errors or warnings.

Linking is the final step and it combines the program object code with other object codes to produce the executable file. The other object codes come from *run-time libraries*, other libraries, or object files that the programmer has created. Finally it saves the executable code as a file on the disk. If any linker errors are encountered the executable file will not be generated.

2 – Introduction to C Programming

2.1 An Example C Program

In order to see the structure of a C program, it is best to start with a simple program. The following code is a C program which displays the message "Hello, World!" on the screen.

```
/* Program-2.1 - My First Program */
#include <stdio.h>
main()
{
     printf("Hello World!");
}
```

The heart of this program is the function **printf**, which actually does the work. The C language is built from functions like **printf** that execute different tasks. The functions must be used in a framework which starts with the word **main()**, followed by the block containing the function(s) which is marked by braces (**{ }**). The **main()** is a special function that is required in every C program.

The first line starting with characters **/ *** and ending with characters *** /** is a *comment*. Comments are used by programmers to add remarks and explanations within the program. It is always a good practice to comment your code whenever possible. Comments are useful in program maintenance. Most of the programs that people write needs to be modified after several months or years. In such cases they may not remember why they have written a program in such a manner. In such cases comments will make a programmer's life easy in understanding the source code and the reasons for writing them. Compiler ignores all the comments and they do not have any effect on the executable program.

Comments are of two types; *single line* comments and *block* comments. Single line comments start with two slashes {**/ /** } and all the text until the end of the line is considered a comment. Block comments start with characters **/ *** and end with characters *** /** . Any text between those characters is considered a block of comments.

Lines that start with a pound (#) symbol are called *directives* for the preprocessor. A directive is not a part of the actual program; it is used as a command to the preprocessor to direct the translation of the program. The directive **# include** appears in all programs as it refers to the *standard input output header* file (**stdio.h**). Here, the header file **stdio.h** includes information about the **printf( )** function. When using more than one directive each must appear on a separate line.

A header file includes data types, macros, function prototypes, inline functions and other common declarations. A header file does not include any implementation of the functions declared. The C preprocessor is used to insert the function definitions into the source files and the actual library file which contains the function implementation is linked at link time. There are prewritten libraries of functions such as **printf()** to help us. Some of these functions are very complex and long. Use of prewritten functions makes the programmers life easier and they also allow faster and error free development (since functions are used and tested by many programmers) development.

The function **printf** is embedded into a statement whose end is marked by a semicolon (**;**). Semicolon indicates the end of a statement to the compiler.

The C language is case sensitive and all C programs are generally written in lowercase letters. Some of the special words may be written in uppercase letters.

2.2 Your First C Program

Let us write our first real C program. Throughout this course students will be using Linux as the development platform. A C program written for the Linux platform (or for UNIX) is slightly different to the program shown earlier. A modified version of the Hello World program is given next.

```
/* Program-2.2 - My First Program */
#include <stdio.h>
int main()
{
      printf("Hello World!");
      return 0;
}
```

Similar to the previous program, **main()** is placed as the first function; however with a slight difference. The *keyword* **int** is added before the function **main**. The keyword **int** indicates that **main()** function *returns* an integer value. Most functions return some value and sometimes this return value indicates the success or the failure of a function.

Because function **main()** returns an integer value, there must be a statement that indicates what this value is. The statement "**return 0;**" does that; in this case it returns zero (conventionally 0 is returned to indicate the success of a function).

Use your favorite text editor in Linux and type the above program. Save it as **HelloWorld.c** (all C source files are saved with the extension **.c**). Then use the shell and go to the directory that you have saved your source file. Use the command **gcc HelloWorld.c** to compiler your program. If there are no errors an executable file will be created in the same location with the default file name **a.out**. To execute the program at the prompt type **./ a.out** (i.e. to execute the **a.out** file in current directory). When you execute your program it should display "Hello World!" on the screen.

```
$ vim HelloWorld.c
$ gcc HelloWorld.c
$ ./a.out
Hello World!$
```

However you will see the prompt (**$**) appearing just after the message "**Hello World!**" you can avoid this by adding the *new line* character (**\ n**) to the end of the message. Then your modified program should look like the following:

```
/* Program-2.3 - My First Program */
#include <stdio.h>
int main()
{
      printf("Hello World!\n");
      return 0;
}
```

2.3 C Tokens

The smallest element identified by the compiler in a source file is called a *token.* It may be a single character or a sequence of characters to form a single item. Tokens can be classified as *keywords*, *literals*, *identifiers*, *operators*, etc. Literals can be further classified as numeric constants, character constants and string constants.

Language specific tokens used by a programming language are called *keywords*. Keywords are also called as *reserved words*. They are defined as a part of the programming language therefore cannot be used for anything else. Any user defined literals or identifiers should not conflict with keywords or compiler directives. Table 2.1 lists keywords supported by the C language.

Table 2.1 – The C language keywords

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

Literals are factual data represented in a language. Numeric constants are an uninterrupted sequence of digits (possibly containing a period). Numerical values such as 123, 10000 and 99.99 are examples. Character constants represents a single character and it is surrounded by single quotation mark (**'**). Characters such as 'a', 'A', '$' and '4' are examples. A sequence of characters surrounded by double quotation marks (inverted comma **""**) is called a *string* constant. A statement such as "I like ice cream." is a string constant.

Identifiers are also referred as names. A valid identifier is composed of a letter followed by a sequence of letters, digits or underscore (_) symbols. An identifier must begin with a letter and the rest can be letters, digits or underscores. Identifies are case sensitive; therefore the identifier **abc** is different from **ABC** or **Abc**.

C identifiers can be very long and so it allows descriptive names like "number_of_students" and "Total_number_of_cars_produced_per_year". Sometimes a C compiler may consider only the first 32 characters in an identifier. While defining identifiers programmers should follow some of the naming standards for better readability of the program. One such standard is the use of underscores symbol (_) to combine two words (example: sub_total).

Operators are used with operands to build expressions. For example "**4+5**" is an expression containing two operands (**4** and **5**) and one operator (**+** symbol). C supports large number of mathematical and logical operators such as **+** , **-**, ***** , **/** , **%** , **^** , **&**, **&&** , **|** , **||** , etc. Operators will be discussed in chapter 3.

The C language uses several symbols such as semicolons (**;**), colons (**:**), commas (**,**), apostrophes (**'**), quotation marks (**""**), braces (**[]**), brackets (**{ }**), and parentheses (**()**) to group block of code as a single unit.

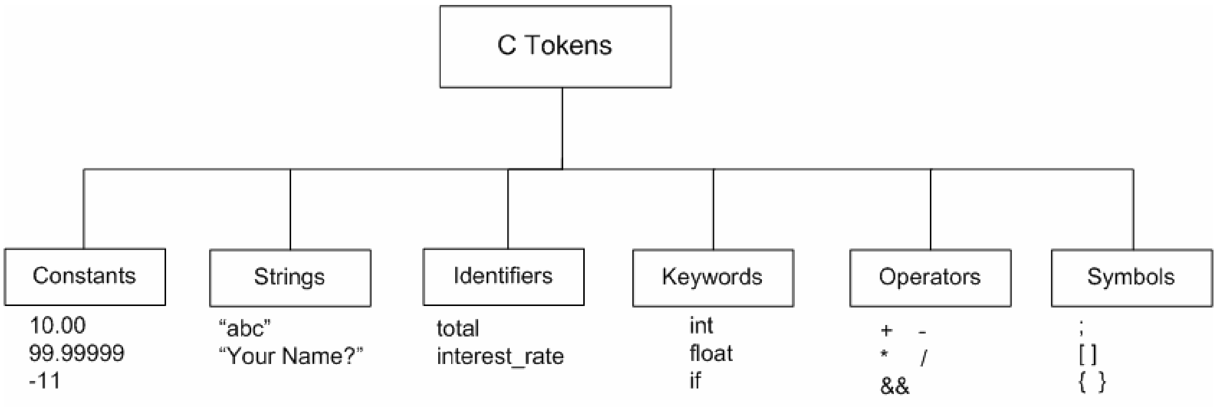


Figure 2.1 – C Tokens

2.4 Displaying Text

The **printf** is the most important function used to display text on the screen. It has two parentheses which contains the string to be displayed, enclosed in quotation marks. Consider Program-2.4 given below. It displays two successive statements using the **printf** function.

```
/* Program-2.4 */
#include <stdio.h>
int main()
{
      printf("Hi there");
      printf("How are you?");
      return 0;
}
```

If you compile this program and run it, the displayed output is:

```
Hi thereHow are you? $
```

How does this happens? The **printf** function first prints the string "Hi there"; the second time **printf** function starts printing the second string from next position on the screen. This can be modified by adding the new line (**\ n**) character before the start of the second string or at the end of the first string. Therefore the modified program can either be Program-2.5a or Program-2.5b.

```
/* Program-2.5a */
#include <stdio.h>
int main()
{
      printf("Hi there ");
      printf("\nHow are you?");
      return 0;
}

/* Program-2.5b */
#include <stdio.h>
int main()
{
      printf("Hi there\n");
      printf("How are you?");
      return 0;
}
```

Now the output is

```
Hi there
How are you?$
```

By adding another **\ n** at then end of the second string you can move the prompt (**$**) to the next line.

Exercise 2.1: Write a C program to display the following text on the screen.

```
University of Moratuwa
Katubedda,
Moratuwa,
Sri Lanka
----------------------
www.mrt.ac.lk
```

2.4.1 Escape Codes

Escape codes are special characters that cannot be expressed otherwise in the source code such as new line, tab and single quotes. All of these characters or symbols are preceded by an inverted (back) slash (****). List of such escape codes are given in Table 2.2. Note that each of these escape codes represents one character, although they consists of two characters.

Table 2.2 – Escape codes

Escape Code	**Meaning**
\a	Audible alert (bell)
\b	Back space
\f	Form feed
\n	New line
\t	Horizontal tab
\v	Vertical tab
\’	Single quote (‘)
\”	Double quote (“)
\?	Question mark (?)
\\	Backslash (\)

2.5 Data Types

A data type in a programming language is a set of data with values having predefined characteristics such as integers and characters. The language usually specifies the range of values for a given data type, how the values are processed by the computer and how they are stored. Storage representations and machine instructions to handle data types differ form machine to machine.

The variety of data types available allows the programmer to select the type appropriate to the needs of the application as well as the machine. C supports a number of data types; if they are not enough programmers can also define their own data types. C supports three classes of data types:

1. Primitive (or basic) data types – these are the fundamental data types supported by the language. These can be classified as integer types, floating point types and character types.
2. User defined data types – based on the fundamental data types users can define their own data types. These include type defined data types (using **typedef** keyword) and enumerated types (using **enum** keyword).
3. Derived data types – programmers can derive data types such as arrays, structures, unions and pointers by combining several data types together.

2.5.1 Primitive Data Types

The C language supports five primitive data types; namely integers (**int**) floating point numbers (**float**), double precision floating point numbers (**double**), characters (**char**) and void (**void**). Many of these data types can be further extended as **long int** and **long double**. Each of these data types requires different storage capacities and has different range of values depending on the hardware (see Table 2.3). Character (**char**) type is considered as an integer type and actual characters are represented based on their ASCII value.

Table 2.3 – Basic C data types (on a 32-bit machine)

Data Type	**Size in Bits**	**Range of values**
char	8	-128 to +127
int	32	-2147483648 to 2147483647
float	32	3.4e-38 to 3.4e+38 (accuracy up to 7 digits)
double	64	1.7e-308 to 1.7e+308 (accuracy up to 15 digits)
void	0	Without value (null)

Table 2.4 – Basic C data types and modifiers (on a 32-bit machine)

Data Type	**Size in Bits**	**Range of values**
Char	8	-128 to +127
unsigned char	8	0 to 255
signed char	8	-128 to +127
int	32	-2147483648 to +2147483647
signed int	32	-2147483648 to +2147483647
unsigned int	32	0 to 4294967295
short	8	-128 to +127
short int	8	-128 to +127
signed short int	8	-128 to +127
unsigned short int	8	0 to 255
long	32	-2,147,483,648 to 2,147,483,647
long int	32	-2,147,483,648 to 2,147,483,647
unsigned long	32	0 to 4,294,967,295
signed long	32	-2,147,483,648 to 2,147,483,647
float	32	3.4e-38 to 3.4e+38
double	64	1.7e-308 to 1.7e+308 (accuracy up to 15 digits)
long double	80	3.4e-4932 to 1.1e+4932 (accuracy up to 19 digits)

2.5.2 Modifiers

The basic data types can be modified by adding special keywords called *data type modifiers* to produce new features or new types. The modifiers are **signed**, **unsigned**, **long** and **short**. For example **short int** represents fairly small integer values and require half the amount of storage as regular **int** numbers. Except with **void** type, the modifiers can be used with all the basic data types as shown in Table 2.4.

2.6 Variables

A variable has a value that can change. It is a memory location that can hold a value of a certain data type. Programmers refer to a variable by its name (identifier) so that it can be accessed during the course of the program. Programmers cannot use any of the keywords as variable names.

2.6.1 Declaring Variables

In order to use a variable in C, the programmer must first declare it specifying the data type. The most important restriction on using a variable in C is that they have to be declared at the beginning of the program. The syntax to declare a new variable is to first write the data type then followed by a valid variable identifier as given in the following examples:

```
int a;
float total;
unsigned int index_no;
short int number_of_students;
```

Above set of expressions declared, variables "**a**" as an integer, "**total**" as a floating point number, "**index_no**" as unsigned integer (since there are no negative index numbers) and "**number_of_students**" as a short integer.

Multiple variables belonging to the same data type can be defined as separate set of expressions or by listing variable names one after the other (should be separated by a coma sign (**,**)).Consider the following examples:

```
int a;
int b;
float total;
float sub_total;
```

above variables can also be defined as:

```
int a,b;
float total, sub_total;
```

After declaring a variable it should be initialised with a suitable value. In C, an uninitialised variable can contain any garbage value therefore the programmer must make sure all the variables are initialised before using them in any of the expressions. Initialising a variable can be done while declaring it, just after declaring it or later within the code (before accessing/evaluating its value within an expression). In initialising a variable any of the following three approaches are valid:

```
int a;
a = 10;
```

or

```
int a=10;
```

or

```
int a;
--------
--------
a = 10;
```

2.6.2 Constants

The value of a constant cannot be changed after an initial value is assigned to it. The C language supports two types of constants; namely *declared constants* and *defined constants*. Declared constants are more common and they are defined using the keyword **const**. With the **const** prefix the programmer can declare constants with a specific data type exactly as it is done with variables.

```
const float pi = 3.141;
```

Programmers can define their own names for constants which are used quite often in a program. Without having to refer to a variable such a constant can be defined simply by using the **#define** pre-processor directive. These are called defined constants. Following expression illustrates the use of the **# define** pre-processor directive

```
#define pi 3.141
```

2.7 Displaying Numbers

When displaying numbers special care must be given to the data type. Each data type has to be used with **printf** function in a specific format. In order to display the correct values using the **printf** function *conversion specifiers* should be used. They are used to instruct the compiler about the type of numbers appearing in the program, which in turn determines the suitable memory storage locations.

Table 2.5 – The printf conversion specifiers

Conversion Specifiers	**Meaning of the output format**
%c	Character
%d	Decimal integer
%e or %E	Scientific notation
%f	Floating point
%g or %G	Scientific notation or floating point (whichever shorter)
%i	Decimal integer
%o	Octal number
%p	Pointer
%s	String of characters
%u	Unsigned decimal integer
%x or %X	Hexadecimal number
%%	Display the % sign

The conversion specifiers are also referred as *format characters* or *format specifiers*. Table 2.5 summarises conversion specifies supported by the **printf** function. Consider the example given below:

```
/* Program-2.6 */
#include <stdio.h>
int main()
{
      printf("%d\n",128);
      printf("%f\n",128.0);
      return 0;
}
```

Executing above program will display the following:

```
128
128.000000
```

The first number is of the type integer while the second number is of the type float. Conversion specifier **% d** stands for decimal while **% f** stands for float. Incorrect use of format characters would result in wrong outputs.

Exercise 2.2: Rewrite Program-2.3 and try swapping the %d and %f. Run your program and observe the output.

Exercise 2.3: Identify the output of each printf function call given in Program-2.7. Execute the program and observe the outputs.

Consider Program-2.8 which makes use of variables. In this program variables “**a**” and “**b**” are initialised with values 10.0 and 3 and the answer **(b/ a)** is stored in variable “**c**”.

```
/* Program-2.7 */
#include <stdio.h>
int main()
{
      printf("%d\n",65*2);
      printf("%d\n",65/2);
      printf("%f\n",65.0/2);
      printf("%c\n",65);
      printf("%x\n",255);
      return 0;
}
```

```
/* Program-2.8 */
#include <stdio.h>
int main()
{
      int a = 3;
      float b = 10.0;
      float c;
      c = b/a;
      printf("A is %d\n",a);         //decimal value
      printf("B is %d\n",b);         //decimal value
      printf("Answer is %f\n",c);    //floating point value
      return 0;
}
```

Executing Program-2.8 will display the following:

```
A is 3
B is 0
Answer is 3.333333
```

In Program-2.8 you may wish to see the answer appearing in a more manageable form like 3.3 or 3.33 rather than 3.333333. This can be achieved by using modifiers along with the format characters in order to specify the required field width.

The format **`% .0f`** will suppress all the digits to the right of the decimal point, while the format **`% .2f`** will display first two digits after that decimal point.

Exercise2.4: Modify Program-2.8 so that it displays an answer which is correct up to 2 decimal points.

Exercise 2.5: Write a program to assign the number 34.5678 to a variable named “number” then display the number rounded to the nearest integer value and next the number rounded to two decimal places.

Program-2.9 illustrates the use of character variables.

```
/* Program-2.9 */
#include <stdio.h>
int main()
{
      char first_letter;
      first_letter = ‘A’;
      printf("Character %c\n", first_letter);   //display character
      printf("ASCII value %d\n", first_letter); //display ASCII value
      return 0;
}
```

Executing Program-2.9 will display the following:

```
Character A
ASCII value 65
```

2.8 Formatted Input

We have already used **printf** function to display formatted output. The **scanf** is a similar function that is used to read data into a program. The **scanf** function accepts formatted input from the keyboard. Program-2.10 demonstrates the use of the **scanf** function:

```
/* Program-2.10 */
#include <stdio.h>
int main()
{
      float a,b,sum;
      scanf("%f", &a);              // read 1st number
      scanf("%f", &b);              // read 2nd number
      sum = a + b;                  // total
      printf("a+b = %f\n", sum);    //display answer as float
      return 0;
}
```

Three variables are declared (**a**, **b** and **sum**) in Program-2.10. Values of "**a**" and "**b**" are to be read from the keyboard using the **scanf** function, while the value of the variable "**sum**" is calculated as the summation of "**a**" and "**b**". The **scanf** function uses the same set of formatting characters as the **printf** function to describe the type of expected input.

Note that the **scanf** function uses the variables "**a**" and "**b**"as "**&a**" and "**&b**". The symbol "**&**" is called the *address of operator*. The string "**&a**" represents the memory address containing variable "a" and is called a *pointer* (see Section 8).

When the program executes, it waits for the user to type the value of "**a**" followed by the **Enter** key and then it waits for the value of "**b**" followed by another the **Enter** key. The supplied input values can be separated either by pressing **Enter** key or by leaving a blank space between the numbers. Both of the following inputs are valid for Program-2.10.

```
2
3
Answer > 5.000000
```

or

```
2 3
Answer > 5.000000
```

Exercise 2.6: Modify Program-2.10 so that it displays the answer with only two decimal points.

The **scanf** function also supports mixed types of input data. Consider the following line of code:

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

The **scanf** function accepts variable "**a**" as an integer and "**b**" as a floating point number.

In certain cases you may want to separate your inputs using a comma (,) rather than using the **Enter** key or a blank space. In such cases you must include the comma between format characters as:

```
scanf("%f,%f", &a,&b);
```

For such a program your inputs should be in the form:

```
2,3
```

One deficiency of the **scanf** function is that it cannot display strings while waiting for user input. This will require use of an additional function like the **printf** in order to display a message as a prompt for the user reminding the required data item.

Exercise 2.7: Modify Program-2.10 so that it displays the following output when executed.

```
Enter 1st number : 2
Enter 2nd number : 3
Answer > 5
```

3 – Operators in C

Expressions can be built up from literals, variables and operators. The operators define how the variables and literals in the expression will be manipulated. C supports several types of operators and they can be classified as:

- Assignment operator
- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Special operators
- Pointer operators

Pointer operators and special operators will not be covered in this chapter. They will be introduced later under relevant sections.

.

3.1 Assignment Operator

The assignment operator is the simple equal sign (=). The assignment operator is used to assign a value to a variable. The format of an assignment statement is:

variable-name = expression;

The expression can be a single variable or a literal, or it may contain variables, literals and operators. The assignment operation always takes place from right to left. Consider the following set of examples:

```
a = 5;      // value of variable ‘a’ becomes 5
a = 5+10;   // value of variable ‘a’ becomes 15
a = 5 + b;  // value of variable ‘a’ becomes 5 + value of b
a = b + c;  // value of variable ‘a’ becomes value of b + value of c
a = (x*x + y*y)/2;
```

In C lots of shortcuts are possible. For example, instead of the statement,

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

the programmer may use the shorthand format:

```
a += b;
```

Such operators are called *compound assignment operators*. The assignment operator can be combined with the major arithmetic operations such as; +, -, *, / and %. Therefore many similar assignments can be used such as:

```
a -= b;     // is same as     a = a-b;
a *= b;     // is same as     a = a*b;
a /= b;     // is same as     a = a/b;
a %= b;     // is same as     a = a%b;
```

Table 3.1 – Arithmetic operators

Operator	**Action**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division
++	Increment (extended)
--	Decrement (extended)

3.2 Arithmetic Operators

C supports five major arithmetic operators and two extended (shortcuts) operators (Table 3.1). Program-3.1 illustrates the usage of major arithmetic operators.

```
/* Program-3.1 */
#include <stdio.h>
int main()
{
   int a,b;
   printf("Enter a: ");
   scanf("%d", &a);                 //read value of a
   printf("Enter b: ");
   scanf("%d", &b);                 //read value of b

   printf("\na+b = %d", a+b);       //display sum of a & b
   printf("\na-b = %d", a-b);       //display subtraction of b from a
   printf("\na*b = %d", a*b);       //display multiplication of a & b
   printf("\na/b = %d", a/b);       //display division of a by b
   printf("\na%%b = %d", a%b);    //display modulus of a divided by b

   return 0;
}
```

Executing Program-3.1 with "**a**" as 5 and "**b**" as 2 will display the following:

```
Enter a: 5
Enter b: 2

a+b = 7
a-b = 3
a*b = 10
a/b = 2
a%b = 1
```

3.2.1 Increment and Decrement Operators

Increment and decrement operators are very useful operators. The operator **++** means "add 1" or "increment by 1". Similarly operator **--** mean "subtract 1" or "decrement by 1". They are also equivalent to **+=1** and **-=1.** For example, instead of using the following expression to increment a variable:

```
a = a + 1;
```

you may use:

```
a +=1;       or     a++;
```

Also expression:

```
a = a - 1;
```

can be written as:

```
 a -=1;      or     a--;
```

Both increment and decrement operators can be used as a *prefix* or as a *suffix*. The operator can be written before the identifier as a prefix (**++a**) or after the identifier as a suffix (**a++**). In simple operations such as **a++** or **++a** both have exactly the same meaning. However in some cases there is a difference. Consider the following set of statements:

```
int a, x;
a = 3;
x = ++a;
```

After executing above segment of code, "**a**" will be **4** and "**x**" will be **4**.

In line 3, first the variable "**a**" is incremented before assigning it to variable "**x**". Consider the following segment of code.

```
int a, x;
a = 3;
x = a++;
```

After executing above code segment "**a**" will be **4** and "**x**" will be **3**.

In the second approach in line 3 first "**a**" is assigned to "**x**" and then "**a**" is incremented.

Exercise 3.1 **–** Predict the values of variables "a", "b", "sum1" and "sum2" if the following code segment is executed.

```
int a, b;
a = b = 2;
sum1 = a + (++b);
sum2 = a + (b++);
```

3.2.2 Precedence and Associatively of Arithmetic Operators

Precedence defines the priority of an operator while associativity indicates which variable(s) an operator is associated with or applied to. Table 3.2 illustrates the *precedence* of arithmetic operators and their *associativity*.

Table 3.2 – Precedence of arithmetic operators

Operator	**Precedence**	**Associativity**
++, --	Highest	Right to left
* / %	↓	Left to right
+ –	Lowest	Left to right

In the same expression, if two operators of the same precedence are found, they are evaluated from left to right, except for increment and decrement operators which are evaluated from right to left. Parentheses can also be used to change the order of evaluation and it has the highest precedence. It is recommended to use parentheses to simplify expressions without depending on the precedence.

3.3 Relational and Logical Operators

The relational operators are used to compare values forming relational expressions. The logical operators are used to connect relational expressions together using the rules of formal logic. Both types of expressions produce TRUE or FALSE results. In C, FALSE is the zero while any nonzero number is TRUE. However, the logical and relational expressions produce the value "1" for TRUE and the value "0" for FALSE. Table 3.3 shows relational and logical operators used in C.

Table 3.3 – Relational and logical operators

Operator	**Action**
Relational Operators	
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!=	Not equal
Logical Operators	
&&	AND
||	OR
!	NOT

3.3.1 Precedence of Relational and Logical Operators

As arithmetic operators, relational and logical operators also have precedence. Table 3.4 summarises the relative precedence of the relational and logical operators. These operators are lower in precedence than arithmetic operators.

Table 3.4 – Precedence of relational and logical operators

Operator	**Precedence**	**Associativity**
!	Highest	Right to left
> >= < <=	↓	Left to right
= = !=	↓	Left to right
&&	↓	Left to right
||	Lowest	Left to right

For an example consider the following expression:

```
10 > 8+1
```

is equivalent to the expression:

```
10 > (8+1)
```

In order to understand how logical expressions are evaluated consider the following expression:

```
a==b && c > d
```

This expression says that; "(**a** is equivalent to **b**) **AND** (**c** is greater than **d**)". In other words this expression is evaluated as TRUE if both the conditions are met. That is if **a = = b** is TRUE and **c > d** is also TRUE. If AND (**&&**) is replaced by logical OR (**||**) operation then only one condition needs to be TRUE.

Consider another expression (assume both "a" and "b" are integers and their value is 5):

```
!(a==b)
```

In this case, since both "**a**" and "**b**" have similar value 5, the logical value of the expression within the parenthesis is TRUE. However the NOT operation will negate the result which is TRUE. Therefore the final result will be FALSE.

In order to see the effect of precedence consider the following expression:

```
a==b && x==y || m==n
```

Since equal operator (**= =**) has a higher precedence than the logical operators the equality will be checked first. Then the logical AND operation will be performed and finally logical OR operation will be performed. Therefore this statement can be rewritten as:

```
((a==b) && (x==y)) || (m==n)
```

This expression is evaluated as TRUE if:

- **a= = b** is evaluated as TRUE and **x= = y** is evaluated as TRUE
- Or **m= = n** is evaluated as TRUE.

3.4 Bitwise Operators

Using C you can access and manipulate bits in variables, allowing you to perform low level operations. C supports six bitwise operators and they are listed in Table 3.5. Their precedence is lower than arithmetic, relational and logical operators (see Table 3.6).

Table 3.5 – Bitwise operators

Operator	**Action**
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	One's complement
>>	Right shift
<<	Left shift

Table 3.6 – Precedence of bitwise operators

Operator	**Precedence**	**Associativity**
~	Highest	Left to right
<< >>	↓	Left to right
&	↓	Left to right
^	↓	Left to right
|	Lowest	Left to right

Bitwise AND, OR and XOR operators perform logical AND, OR and XOR operations at the bit level. Consider following set of expressions:

```
int a,b,c;
a = 12;
b = 8;
c = a & b;
```

The bitwise AND operation will be performed at the bit level as follows:

```
a = 12 → 00001100
b =  8 → 00001000 &
         00001000
```

Then the variable "**c**" will hold the result of bitwise AND operation between "**a**" and "**b**" which is 00001000_2 (8_{10}).

Example 3.1 – Write a C program to convert a given character from uppercase to lowercase and vice versa.

ASCII values of the uppercase and lowercase characters have a difference of 32. For example, in ASCII, "A" is represented by 65_{10} while "a" is represented by 97_{10} (97-65 = 32). At the bit level, only difference between the two characters is the 5th bit.

```
65 = 0 1 0 0 0 0 0 1₂
97 = 0 1 1 0 0 0 0 1₂
32 = 0 0 1 0 0 0 0 0₂
```

Therefore by inverting the 5th bit of a character it can be changed from uppercase to lowercase and vice versa. Bitwise XOR operation can be used to invert bits. Therefore any ASCII value XOR with 32 will invert its case form upper to lower and lower to upper. This is the concept used in Program 3.2 which converts a given character from uppercase to lowercase and vice versa.

```
/* Program-3.2 */
#include <stdio.h>
int main()
{
   char input;
   printf("Character to convert: ");
   scanf("%c",&input);                    //read character
   printf("Converted character: %c", input ^ 32); //input XOR 32

   return 0;
}
```

Executing Program-3.2 with character "b" as the input will display the following:

```
Character to convert: b
Converted character: B
```

Executing Program-3.2 with character "Q" as the input will display the following:

```
Character to convert : Q
Converted character : q
```

3.4.1 Shift Operators

Shift operators allow shifting of bits either to left or right. We know that 8 is represented in binary as:

```
8 = 00001000
```

and when it is shifted by one bit to right, we get:

```
8 = 00001000 → 00000100 = 4
```

Similarly when the number is shifted by one bit to left, we get:

```
16 = 00010000 ← 00001000 = 8
```

The C right shift (**> >**) operator shifts an 8-bit number by one or more bits to right while left shift (**< <**) operator shifts bits to left. Shifting a binary number to left by one bit will multiply it by 2 while shifting it to right by one bit will divide it by 2. After the shift operation the final result should also be a 8-bit number, therefore any additional bits are removed and missing bits will be filled with zero.

In general the right shift operator is used in the form:

```
variable >> number-of-bits
```

and left shift is given in the form:

```
variable >> number-of-bits
```

In the following expression the value in the variable "a" is shift by 2 bits to right:

```
a >> 2
```

Table 3.7 summarises all the operators supported by C, their precedences and associativities. However some operators can be confusing, as they use the same characters (for example, ***** is used for multiplication and also to denote pointers). However the following rule will reduce any confusion. The *unary operators* (operators with only one operand such as **&a** (address of a)) have a higher precedence than the *binary operators* (operators with two operands such as **a* b**). When operators have the same precedence an expression is evaluated left to right.

Table 3.7 – Precedence of C operators

Operator	**Precedence**	**Associativity**
( ) [ ] -> .	Highest	Left to right
! ~ ++ --	↓	
& *	↓	Right to left
* / %	↓	Left to right
+ -	↓	Left to right
<< >>	↓	Left to right
< <= > >=	↓	Left to right
= = !=	↓	Left to right
&	↓	Left to right
^	↓	Left to right
|	↓	Left to right
&&	↓	Left to right
||	↓	Left to right
? :	↓	Left to right
= += -= *= /=	↓	Right to left
,	Lowest	Left to right

4 – Conditional Control Structures

A Program is usually not limited to a linear sequence of instructions. In real life, a programme usually needs to change the sequence of execution according to some conditions. In C, there are many *control structures* that are used to handle conditions and the resultant decisions. This chapter introduces **if-else** and **switch** constructs.

4.1 The if Statement

A simple condition is expressed in the form:

```
if (condition)
      statement;
```

It starts with the keyword **if**, followed by a condition (a logical expression) enclosed within parenthesis, followed by the result statement. The resulting statement is executed if the condition is evaluated as TRUE. Note that there is no semicolon (;) after the condition expression. Consider the following example:

```
if (marks >50)
      printf("You have passed the exam!");
```

If the value of the variable "**marks**" is greater than **50**, the message "**You have passed the exam!**" is displayed on the screen; otherwise the statement is skipped and no message is displayed. Program 4.1 illustrates the use of this in a C program.

```
/* Program-4.1 */
#include <stdio.h>
int main()
{
   int marks;
   printf("Enter marks: ");
   scanf("%d", &marks);             //get marks

   if (marks >50)                   // if marks >50 display message
      printf("You have passed the exam!");

   return 0;
}
```

Executing Program-4.1 with different inputs will behave as follows:

Case 1: `Enter marks: 73`

```
You have passed the exam!
```

Case 2: `Enter marks: 34`

Case 3: `Enter marks: 50`

In the second and third cases, the message "**You have passed the exam!**" will not be displayed.

More than one statement can be executed as a result of the condition by embedding set of statements in a block (between two braces **{ }**).

Example 4.1 – Write a program which accepts a number (an amount of money to be paid by a customer in rupees) entered from the keyboard. If the amount is greater than or equal to 1000 rupees, a 5% discount is given to the customer. Then display the final amount that the customer has to pay.

First the program needs to check whether the given amount is greater than or equal to 1000; if it is the case a 5% discount should be given. Then the final amount needs to be displayed. All these are done only if the condition is TRUE. So instructions which compute discount and final amount should be executed as a block. Program 4.2 is an implementation of this.

```
/* Program-4.2 */
#include <stdio.h>
int main()
{
   float amount,final_amount, discount;
   printf("Enter amount: ");
   scanf("%f", &amount);      //get amount

   if (amount >= 1000)        // if amount >= 1000 give discount
   {
      discount = amount* 0.05;
      final_amount = amount - discount;
      printf ("Discount: %.2f", discount);
      printf ("\nTotal: %.2f", final_amount);
   }

   return 0;
}
```

Executing Program-4.2 with 1250.25 as the keyboard input display the following:

```
Enter amount: 1250.25
Discount: 62.51
Total: 1187.74
```

In Program-4.1 if the condition is TRUE, the set of statements inside the block are executed. If the condition is FALSE (if the amount is less than 1000) those statements will not be executed.

Example 4.2 – Modify Program-4.2 so that it displays the message "No discount…" if the amount is less than 1000.

Another **if** clause is required to check whether the amount is less than 1000. The second **if** clause can be used before or after the first (existing) **if** clause. Program-4.3 below has been modified from Program-4.2 to address this.

```
/* Program-4.3 */
#include <stdio.h>
int main()
{
   float amount,final_amount, discount;
   printf("Enter amount: ");
   scanf("%f", &amount);      //get amount

   if (amount >= 1000)        // if amount >= 1000 give discount
   {
      discount = amount* 0.05;
      final_amount = amount - discount;
      printf ("Discount: %.2f", discount);
      printf ("\nTotal: %.2f", final_amount);
   }
   if (amount < 1000)         // if amount < 1000 no discount
      printf ("No discount...");
   return 0;
}
```

Exercise 4.1 – Modify Program-4.1 so that it displays the message "You are failed!", if marks are less than or equal to 50.

In many programs it is required to perform some action when a condition is TRUE and another action when it is FALSE. In such cases, the **if** clause can be used to check the TRUE condition and act upon it. However it does not act upon the FALSE condition. Therefore the expression resulting the FALSE condition needs to be reorganised. In example 4.2 when we wanted to identify the case where the amount is not greater than 1000 (the FALSE case) we were checking whether it is less than 1000 (**< 1000**).

The C language allows us to use the **if-else** structure in such scenarios. You can include both the cases (TRUE and FALSE) using the **if-else** structure.

4.2 The if-else Structure

The if-else structure takes the form:

```
if (condition)
      statement-1;
else
      statement-2;
```

When the condition is evaluated, one of the two statements will be executed and then the program resumes its original flow. Blocks make it possible to use many statements rather than just one. Then the Program-4.3 can be modified as follows:

```
/* Program-4.4 */
#include <stdio.h>
int main()
{
   float amount,final_amount, discount;
   printf("Enter amount: ");
   scanf("%f", &amount);      //get amount

   if (amount >= 1000)        // if amount >= 1000 give discount
   {
      discount = amount* 0.05;
      final_amount = amount - discount;
      printf ("Discount: %.2f", discount);
      printf ("\nTotal: %.2f", final_amount);
   }
   else                       // else no discount
      printf ("No discount...");
   return 0;
}
```

Exercise 4.2 – Modify Program-4.1 so that it displays the message "You have passed the exam!", if marks are greater than 50. If not display the message "You have failed!".

Exercise 4.3 – Write a program to identify whether a number input from the keyboard is even or odd. If it is even, the program should display the message "Number is even", else it should display "Number is odd".

4.3 The if-else-if Ladder

In certain cases multiple conditions are to be detected. In such cases the conditions and their associated statements can be arranged in a construct that takes the form:

```
if (condition-1)
      statement-1;
else if (condition-2)
      statement-2;
else if (condition-3)
      statement-3;
…
else
      statement-n;
```

The above construct is referred as the *if-else-if* ladder. The different conditions are evaluated starting from the top of the ladder and whenever a condition is evaluated as TRUE, the corresponding statement(s) are executed and the rest of the construct it skipped.

Example 4.3 – Write a program to display the student's grade based on the following table:

Marks	Grade
>=75	A
> 50 and <75	B
> 25 and <50	C
< 25	F

In this case multiple conditions are to be checked. Marks obtained by a student can only be in one of the ranges. Therefore if-else-if ladder can be used to implement following program.

```
/* Program-4.5 */
#include <stdio.h>
int main()
{
   int marks;
   printf("Enter marks: ");
   scanf("%d", &marks);                    //read marks

   if(marks > 75)                          // if over 75
      printf("Your grade is: A");
   else if(marks >= 50 && marks <75)       // if between 50 & 75
      printf("Your grade is: B");
   else if(marks >= 25 && marks <50)       // if between 25 & 50
      printf("Your grade is: C");
   else
      printf ("Your grade is: F");         // if less than 25

   return 0;
}
```

Notice that in Program-4.5, some of the conditional expressions inside the **if** clause are not as simple as they were earlier. They combine several expressions with logical operators such as AND (**&&**) and OR (**||**). These are called *compound relational tests*.

Due to the top down execution of the if-else-if ladder Program-4.5 can also be written as follows:

```
/* Program-4.6 */
#include <stdio.h>
int main()
{
   int marks;
   printf("Enter marks: ");
   scanf("%d", &marks);              //get marks

   if(marks > 75)
      printf("Your grade is A");     // if over 75
   else if(marks >= 50 )
      printf("Your grade is B");     // if over 50
   else if(marks >= 25 )
      printf("Your grade is C");     // if over 25
   else
      printf ("Your grade is F");    // if not

   return 0;
}
```

In Program-4.6, when the marks are entered from the keyboard the first expression (**marks > 75**) is evaluated. If **marks** is not greater than 75, the next expression is evaluated to see whether it is greater than 50. If the second expression is not satisfied either, the program evaluates the third expression and so on until it finds a TRUE condition. If it cannot find a TRUE expression statement(s) after the **else** keyword will get executed.

4.4 Nesting Conditions

Sometimes we need to check for multiple decisions. This can be accomplished by two approaches; using compound relational tests or using *nested conditions*. When conditions are nested the if-else/if-else-if construct may contain other if-else/if-else-if constructs within themselves. In nesting you must be careful to keep track of different **if**s and corresponding **else**s. Consider the following example:

```
if (a>=2)
   if (b >= 4)
      printf("Result 1");
   else
      printf("Result 2");
```

An **else** matches with the last **if** in the same block. In this example the **else** corrosponds to the second **if**. Therefore if both **a >= 2** AND **b >= 4** are TRUE "**Result 1**" will be displayed; if **a>= 2** is TRUE but if **b >= 4** is FALSE "**Result 2**" will be displayed. If **a >= 2** is FALSE nothing will be displayed. To reduce any confusion braces can be used to simplify the source code. Therefore the above can be rewritten as follows:

```
if (a>=2)
{
   if (b >= 4)
      printf("Result 1");
   else
      printf("Result 2");
}
```

In the above, if **else** is to be associated with the first **if** then we can write as follows:

```
 if (a>=2)
{
   if (b >= 4)
      printf("Result 1");
}
else
   printf("Result 2");
```

Exercise 4.4 – Rewrite the program in Example 4.3 using nested conditions (i.e. using braces '{' and '}'.

Example 4.4 – A car increases it velocity from *u* ms^{-1} to *v* ms^{-1} within *t* seconds. Write a program to calculate the acceleration.

The relationship among acceleration (*a*), *u*, *v* and *t* can be given as $v = u + at$. Therefore the acceleration can be found by the formula $a = \frac{v-u}{t}$. In the program we implement users can input any values for *u*, *v* and *t*. However, to find the correct acceleration *t* has to be non zero and positive (since we cannot go back in time and a number should not be divided by zero). So our program should make sure it accepts only the correct inputs. The Program-4.7 calculates the acceleration given *u*, *v* and *t*.

4.5 Conditional Operator

Conditional operator (**?:**) is one of the special operators supported by the C language. The conditional operator evaluates an expression and returns one of two values based on whether condition is TRUE or FALSE. It has the form:

```
condition ? result-1 : result-2;
```

If the condition is TRUE the expression returns **result-1** and if not it returns **result-2**.

Example 4.5 – Consider the following example which determines the value of variable "b" based on the whether the given input is greater than 50 or not.

```
/* Program-4.7 */
#include <stdio.h>
int main()
{
   float u,v,t,a;
   printf("Enter u (m/s): ");
   scanf("%f", &u);     //get starting velocity
   printf("Enter v (m/s): ");
   scanf("%f", &v);     //get current velocity
   printf("Enter t (s) : ");
   scanf("%f", &t);     //get time

   if(t >= 0)
   {
      a = (v-u)/t;
      printf("acceleration is: %.2f m/s", a);
   }
   else
      printf ("Incorrect time");

   return 0;
}

/* Program-4.8 */
#include <stdio.h>
int main()
{
   int a,b;
   printf("Enter value of a: ");
   scanf("%d", &a);     //get starting velocity
   b = a > 50 ? 1 : 2;
   printf("Value of b: %d", b);
   return 0;
}
```

Executing Program-4.8 display the following:

```
Enter value of a: 51
Value of b: 1

Enter value of a: 45
Value of b: 2
```

If the input is greater than 50 variable “**b**” will be assigned “1” and if not it will be assigned “2”.

Conditional operator can be used to implement simple if-else constructs. However use of conditional operator is not recommended in modern day programming since it may reduce the readability of the source code.

4.6 The switch Construct

Instead of using if-else-if ladder, the switch construct can be used to handle multiple choices, such as menu options. The syntax of the switch construct is different from if-else construct. The objective is to check several possible constant values for an expression. It has the form:

```
switch (control variable)
{
   case constant-1:
      statement(s);
      break;
   case constant-2:
      statement(s);
      break;
      …
```

```
        case constant-n:
            statement(s);
            break;
        default:
            statement(s);
    }
```

The switch construct starts with the **switch** keyword followed by a block which contains the different cases. Switch evaluates the control variable and first checks if its value is equal to **constant-1**; if it is the case, then it executes the statement(s) following that line until it reaches the **break** keyword. When the **break** keyword is found no more cases will be considered and the control is transferred out of the switch structure to next part of the program. If the value of the expression is not equal to **constant-1** it will check the value of **constant-2**. If they are equal it will execute relevant statement(s). This process continuous until it finds a matching value. If a matching value is not found among any **case**s, the statement(s) given after the **default** keyword will be executed.

The control variable of the switch must be of the type **int**, **long** or **char** (any other datatype is not allowed). Also note that the value we specify after **case** keyword must be a constant and cannot be a variable (example: n*2).

Example 4.6 – Write a program to display the following menu on the screen and let the user select a menu item. Based on the user's selection display the category of software that the user selected program belongs to.

```
                  Menu
-----------------------------------
1 – Microsoft Word
2 – Yahoo messenger
3 – AutoCAD
4 – Java Games
-----------------------------------
Enter number of your preference:
```

Program-4.9 implements a solution to the above problem:

```
/* Program-4.9 */
#include <stdio.h>
int main()
{
   int a;
   printf("\t\tMenu");
   printf("\n-----------------------------------");
   printf("\n1 - Microsoft Word");
   printf("\n2 - Yahoo messenger");
   printf("\n3 - AutoCAD");
   printf("\n4 - Java Games");
   printf("\n-----------------------------------");
   printf("\nEnter number of your preference: ");

   scanf("%d",&a);                              //read input

   switch (a)
   {
      case 1:                                   //if input is 1
         printf("\nPersonal Computer Software");
         break;
      case 2:                                   //if input is 2
         printf("\nWeb based Software");
         break;
      case 3:                             //if input is 3
         printf("\nScientific Software");
         break;
      case 4:                             //if input is 4
         printf("\nEmbedded Software");
```

```
            break;
         default:
            printf("\nIncorrect input");
      }
      return 0;
   }
```

Executing Program-4.9 will display the following:

```
              Menu
-----------------------------------
1 - Microsoft Word
2 - Yahoo messenger
3 - AutoCAD
4 - Java Games
-----------------------------------
Enter number of your preference: 1

Personal Computer Software
```

Exercise 4.5 – Develop a simple calculator to accept two floating point numbers from the keyboard. Then display a menu to the user and let him/her select a mathematical operation to be performed on those two numbers. Then display the answer. A sample run of you program should be similar to the following:

```
Enter number 1: 20
Enter number 2: 12
       Mathematical Operation
-----------------------------------
1 - Add
2 - Subtract
3 - Multiply
4 - Divide
-----------------------------------
Enter your preference: 2

Answer : 8.00
```

In certain cases you may need to execute the same statement(s) for several cases of a switch block. In such cases several cases can be grouped together as follows:

switch (x)

```
{
case 1:
case 2:
case 3:
   printf(“Valid input”);
   break;
default:
   printf(“Invalid input”);
   break;
}
```

5 – Control Structures

A Program is usually not limited to a linear sequence of instructions or conditional structures and it is sometimes required to execute a statement or a block of statements repeatedly. These repetitive constructs are called *loops* or *control structures*. The C language supports three constructs; namely **for**, **while** and **do-while** loops. Rest of this chapter introduces these control structures.

5.1 The for loop

The **for** loop construct is used to repeat a statement or block of statements a specified number of times. The general form of a **for** loop is:

```
for (counter-initialization; condition; increment)
      statement(s);
```

The construct includes the initialization of the counter, the condition and the increment. The main function of the **for** loop is to repeat the statement(s) while the condition remains true. The condition should check for a specific value of the counter. In addition it provides ways to initialize the counter and increment (or decrement) the counter. Therefore the **for** loop is designed to perform a repetitive action for a pre-defined number of times. Consider the following example:

```
/* Program-5.1 */
#include <stdio.h>
int main()
{
   int counter;
   for(counter=1; counter <= 5; counter++)   //loop 5 times
   {
      printf("This is a loop\n");
   }

   return 0;
}
```

Execution of program-5.1 displays:

```
This is a loop
This is a loop
This is a loop
This is a loop
This is a loop
```

In the above example, the variable "**counter**" starts with an initial value of "**1**". The second expression inside the parenthesis determines the number of repetitions of the loop. It reads as; "as long as the counter is less than or equal to 5 repeat the statement(s)". The third expression is the incrimination of the counter; it is achieved by the **++** operator. You may also decrement the counter depending on the requirement, but you have to use suitable control expression and an initial value.

In the first round of execution the "**counter**" is set to "**1**". Then the expression "**counter <= 5**" is evaluated. Since the current value of the "**counter**" is "**1**" expression is evaluated as TRUE. Therefore the **printf** function gets executed. Then the "**counter**" is incremented by the **++** operator and now its new value becomes "**2**". At this point the first round of the loop is completed. Then in the second round the expression is evaluated again. Since the "counter" is "2" the expression will be TRUE. Therefore the **printf** function gets executed for the second time. Then the "**counter**" is incremented once more and its new value becomes "**3**". This process continues for another 2 rounds. After a total of five rounds the "**counter**" becomes "**6**". When the expression is evaluated at the beginning of the sixth round the "**counter**" is greater than 5 therefore expression becomes FALSE. Then the loop will terminate and the control is given to rest of the instructions which are outside the loop.

Exercise 5.1 – Write a C program to display all the integers from 100 to 200.

Example 5.1 – Write a program to calculate the sum of all the even numbers up to 100.

First even number is 0 and the last even number is 100. By adding 2 to an even number the next even number can be found. Therefore the counter should be incremented by 2 in each round. Program-5.2 is an implementation of the above requirement.

```
/* Program-5.2 */
#include <stdio.h>
int main()
{
   int counter, sum;
   sum = 0;
   for(counter=0; counter <= 100; (counter += 2)) //increment by 2
   {
      sum += counter;
   }
   printf("Total : %d", sum);
   return 0;
}
```

Exercise 5.2 – Modify Program-5.2 such that it computes the sum of all the odd numbers up to 100.

Exercise 5.3 – Write a program to compute the sum of all integers form 1 to 100.

Exercise 5.4 – Write a program to calculate the factorial of any given positive integer.

Each of the three parts inside the parentheses of the **for** statement is optional. You may omit any of them as long as your program contains the necessary statements to take care of the loop execution. Even the statement(s) inside the loop are optional. You can omit even all the expressions as in the following example:

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

This loop is an infinite one (called an *infinite loop*). It is repeated continuously unless you include a suitable condition inside the loop block to terminate the execution. If there is no such condition, the only thing you can do is to abort the program (a program can be aborted by pressing **Ctrl+ Break**).

Example 5.2 – Write a program to compute the sum of all integers between any given two numbers.

In this program both inputs should be given from the keyboard. Therefore at the time of development both initial value and the final value are not known.

```
/* Program-5.3 */
#include <stdio.h>
int main()
{
   int num1, num2, sum;
   sum=0;
   printf("Enter first number: ");
   scanf("%d", &num1);                   //read num1
   printf("Enter second number: ");
   scanf("%d", &num2);                   //read num2

   for(; num1 <= num2; num1++)
   {
      sum += num1;                       //sum = sum+num1
   }
   printf("Total : %d", sum);

   return 0;
}
```

5.2 The while Loop

The **while** loop construct contains only the condition. The programmer has to take care about the other elements (initialization and incrementing). The general form of the while loop is:

```
while (condition)
{
      statement(s);
{
```

The loop is controlled by the logical expression that appears between the parentheses. The loop continues as long as the expression is TRUE. It will stop when the condition becomes FALSE it will stop. You need to make sure that the expression will stop at some point otherwise it will become an infinite loop. The **while** loop is suitable in cases where the exact number of repetitions is not known in advance. Consider the following example:

```
/* Program-5.4 */
#include <stdio.h>
int main()
{
   int num;
   printf("Enter number of times to repeat: ");
   scanf("%d", &num);
   while (num != 0)
   {
      printf("Hello World!\n");
      num--;
   }

   return 0;
}
```

Execution of Program-5.4 with 5 as the input will display the following:

```
Enter number of times to repeat: 5
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

In Program-5.4 the number of times to loop, depends on the user input. In such cases use of **while** loop is desirable than the **for** loop. In program-5.4 variable "**num**" act as the counter. The conditions inside the **while** loop check whether the counter is not equal to zero (**num != 0**) if so it will execute the **printf** function. Next we need to make sure that the program loops only 5 times. That is achieved by decrementing the counter (**num--**) at each round. If the counter is not decremented the program will loop forever.

Exercise 5.5 – What would happen if we enter -1 as the number of times to loop in Program-5.4? Modify Program-5.4 so that it works only for positive integers.

5.3 The do-while Loop

Another useful loop is the **do-while** loop. The only difference between the **do-while** loop and other loops is that in the **do-while** loop the condition comes after the statement(s). It takes the following form:

```
do
{
      statement(s);
}
while (condition);
```

This means that the statement(s) inside the loop will be executed at least once regardless of the condition being evaluated. Consider the following example:

```
/* Program-5.5 */
#include <stdio.h>
int main()
{
   float price, total;
   total = 0 ;                 //set initial value to 0
   do                          //request for price
   {
      printf("Enter price (0 to end): ");
      scanf("%f", &price);     //get price
      total += price;
   } while (price > 0);        // if valid price continue loop

   printf("Total : %.2f", total);

   return 0;
}
```

Execution of Program-5.5 with some inputs will display the following:

```
Enter price (0 to end): 10
Enter price (0 to end): 12.50
Enter price (0 to end): 99.99
Enter price (0 to end): 0
Total : 122.49
```

Program-5.5 accepts prices of items from the keyboard and then it computes the total. User can enter any number of prices and the program will add them up. It will terminate only if zero or any negative number is entered. In order to calculate the total or terminate the program there should be at least one input from the keyboard. Therefore in this type of a program **do-while** loop is recommended than the **while** loop.

Exercise 5.6 – Modify Program-4.8 such that it first displays the menu. Then based on the user selection it should display the correct message. After displaying the correct message it should again display the menu. The program should exit when the user enter 0.

In summary, the **for** loop is recommended for cases where the number of repetitions is known in advance. The **while** loop is recommended for cases where the number of repetitions are unknown or unclear during the development process. The **do-while** loop is recommended for cases where the loop to be executed needs to run at least once regardless of the condition. However, each type of loop can be interchanged with the other two types by including proper control mechanisms.

5.4 Nesting of Loops

Like the conditional structures loops can also be nested inside one another. You can nest loops of any kind inside another to any depth you want. However having large number of nesting will reduce the readability of your source code.

Example 5.3 – Write a C program to display the following pattern:

```
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
```

There are 10 “$” symbols in a single row (10 columns) and there are 5 rows. This can be implemented by having a two loops one nested inside another which print individual “$” symbols. However the **printf** function does not support displaying text on a row that you have already printed. Therefore first you need to fully complete the first row and then you should go to the next. Therefore the loop which handles printing of individual rows should be the outer loop and one which prints elements within a row (columns) should be the inner loop. Program-5.6 displays the above symbol pattern.

```
/* Program-5.6 */
#include <stdio.h>
int main()
{
   int i,j;
   for(i=0;i<=5;i++)          //outer loop
   {
      for(j=0;j<=10;j++)      // inner loop
      {
            printf("$");
      }                       // end of inner loop
      printf("\n");
   }                          //end of outer loop

   return 0;
}
```

Exercise 5.7 – Write a C program to display the following symbol pattern:

```
*
**
***
****
*****
******
```

5.5 The break Keyword

The **break** keyword is used to terminate a loop, immediately bypassing any conditions. The control will be transferred to the first statement following the loop block. If you have nested loops, then the break statement inside one loop transfers the control to the immediate outer loop. The break statement can be used to terminate an infinite loop or to force a loop to end before its normal termination. Consider the following example:

```
/* Program-5.7 */
#include <stdio.h>

int main()
{
   int n;
   for(n=10;n>0;n--)
   {
      printf("Hello World!\n");
      if(n == 5)
      {
         printf("Countdown aborted!");
         break;
      }
   }
   return 0;
}
```

Under normal circumstances the Program-5.7 will display the "**Hello World!**" message 10 times. Notice that in this example the **for** loop is written as a decrement rather than an increment. During the first 5 iterations the program executes normally displaying the message "**Hello World!**". Then at the beginning of the sixth iteration variable "**n**" becomes "**5**". Therefore the **if** condition which evaluates whether "**n==5**" becomes TRUE, so it will execute the **printf** function and then the **break** instruction. At this point the loop will terminate because of the **break** keyword.

Example 5.4 – Write a C program to display the message "Hello World!" 10000 times. The program should allow users to terminate the program at any time by pressing any key before it displays all the 10000 messages.

The C function **kbhit** can be used to check for a keystroke. If a key has been pressed, it returns the value "**1**" otherwise it returns "**0**". The **kbhit** function is defined in the header file **conio.h**.

```
/* Program-5.8 */
#include <stdio.h>
#include <conio.h>

int main()
{
      int i;
      for(i=0;i<=10000;i++)       // loop 10000 times
      {
         printf("Hello World! %d\n",i);
         if(kbhit() != 0)        // if a key is pressed
         {
            printf("Loop terminated");
            break;               //terminate loop
         }
      }
   return 0;
}
```

5.6 The continue Keyword

The keyword **continue** causes the program to skip the rest of the loop in the current iteration, causing it to jump to the next iteration. Consider the following example.

```
/* Program-5.9 */
#include <stdio.h>

int main()
{
   int i;
   for(i=-5;i<=5;i++)       // loop from -5 to 5
   {
      if (i == 0)           // if 0 skip
         continue;
      printf("5 divided by %d is: \t %.2f \n", i, (5.0/i));
   }
   return 0;
}
```

In program-5.9, **5** is divided by all the integers from **-5** to **+** 5. However a number should not be divided by **0**. In Program-5.9, when "**i**" is **0** (when the **if** condition is TRUE) the **continue** keyword is used to skip the rest of the iteration which will skip the **printf** function.

5.7 The exit Function

The **exit** function (defined in the header file **stdlib.h**) is used to terminate the running program with a specific exit code. It takes the following form:

exit (int exit-code)

The exit-code is used by the operating systems and may also be used by the calling program. By convention, an exit-code of **0** indicates a normal exit where as any other value indicates an abnormal exit or an error. If a program is to be terminated before the **return 0;** statement in within the **main** function following code can be used:

```
exit (0);
```

Exercise 5.8 – Write a C program to display a sine table. The program should display all the sine values from 0 to 360 degrees (at 5 degrees increments) and it should display only 20 rows at a time.

6 – Arrays

Arrays are a series of elements of the same data type placed consecutively in memory that can be individually referenced by adding an index to a unique name. Using an array we can store five values of type **int** with a single identifier without having to declare five different variables with a different identifier. Arrays are useful when you store related data items, such as grades received by the students, sine values of a series of angles, etc. Like any other variable in C an array must be declared before it is used. The typical declaration of an array is:

```
data-type array-name[no-of-elements];
```

Notice that the array name must not be separated from the square brackets containing the index. When declaring an array, the number of array elements should be constant. As arrays are blocks of static memory locations of a given size, the compiler must be able to determine exactly how much memory to allocate at compilation time.

6.1 Initialising an Array

An array will not be initialised when it is declared; therefore its contents are undetermined until we store some values in it. The following array can hold marks for five subjects.

```
int marks[5];
```

The elements of an array can be initialised in two ways. In the first approach, the value of each element of the array is listed within two curly brackets (**{ }**) and a comma (**,**) is used to separate one value from another. For example:

```
marks[5] = {55, 33, 86, 81, 67};
```

In the second approach elements of the array can be initialised one at a time. This approach makes use of the format:

```
array-name[index];
```

For example:

```
marks[0] = 55;
marks[1] = 33;
marks[2] = 86;
marks[3] = 81;
marks[4] = 67;
```

In an array index of the first element is considered as zero rather than one. Therefore in an array with "*n*" elements first index is "*0*" and the last index is "*n-1*". This confusion can be overcome by initialising an array with "*n+1*" elements and neglecting the first element (element with zero index). Consider the following example:

```
/* Program-6.1 */
#include <stdio.h>

int main()
{
   int i,sum;
   int marks[5];  //array of 5 elements
   float average;
   sum=0;
   for(i=0;i<5;i++)
   {
      printf("Enter marks for subject %d: ", i+1);
      scanf("%d", &marks[i]);     //get the marks
    }
    for(i=0;i<=5;i++)             //total marks
    {
      sum +=marks[i];
```

```
          }
          average = sum/5.0;    //5.0 indicates a float value
          printf("Average : %.2f", average);
          return 0;
        }
```

Execution of Program-6.1 displays the following:

```
Enter marks for subject 1: 55
Enter marks for subject 2: 33
Enter marks for subject 3: 86
Enter marks for subject 4: 81
Enter marks for subject 5: 67
Average : 64.60
```

The Program-6.1 accepts marks for 5 subjects from the keyboard and stores them in an array named **marks**. Then in the second loop it computes total of all marks stored in the array and finally it computes the average.

6.2 Multidimensional Arrays

The type of arrays that we discussed up to now is called a *one-dimensional* (or single dimensional) array, as it takes one index and store only one type of data. The array that was used in Program-6.1 hold only marks of one student. It can be extended to store marks of many students using a *two-dimensional* array. Such an array is declared in the following form:

```
data-type array-name[size-1][size-2];
```

You can declare an array to hold marks of 100 students with store marks of 5 subjects as in the following example:

```
int students[100][5];
```

The first index defines the number of students and the second index defines the number of subjects. Altogether it declares 500 (100×5) memory locations. Initialising marks of the first student can be performed in the following manner.

```
Marks[0][0] = 55;
marks[0][1] = 33;
marks[0][2] = 86;
marks[0][3] = 81;
marks[0][4] = 67;
```

Similarly we can define arrays with n dimensions and such arrays are called *n-dimensional* or *multidimensional* arrays.

A two-dimensional array is initialised in the same way. The following statement declares and initialises a two-dimensional array of type **int** which holds the scores of three students in five different tests.

```
int students[3][5]= {
                       {55, 33, 86, 81, 67},
                       {45, 46, 86, 30, 47},
                       {39, 82, 59, 57, 60}
                     };
```

Exercise 6.1 – Write a program to store marks of 5 students for 5 subjects given through the keyboard. Calculate the average of each students marks and the average of marks taken by all the students.

7 – Functions

The C functions that you have used so far (such as **printf** and **scanf**) are built into the C libraries, but you can also write your own functions. Therefore functions can be classified as *built-in* and *user defined.* A modular program is usually made up of different functions, each one accomplishing a specific task such as calculating the square root or the factorial. In general a modular program consists of the **main( )** function followed by set of user defined functions as given below:

```
# include ……
# define …..

Prototypes of functions

int main( )
{
  ……….
}

function_1( )
{
  ………..
}

function_2( )
{
  ………..
}

…………

function_n( )
{
  ………..
}
```

The source code contains other elements in addition to the function blocks. It starts with the **# include** directive, followed by the **# define** directive (if any) then followed by the *prototypes* of functions. The prototype is a declaration of a function used in the program. Then comes the program building block which includes the **main( )** function and implementation of the user defined functions.

7.1 A Function

A function is a subprogram that can act on data and return a value. Some functions may accept certain input parameters (such as **printf**), some may return a value of a certain data type (such as **kbhit**) and some may accept as well as return a value (**sqrt**).

Every C program has at least one function, the **main( )**. When a program is executed the **main( )** is called automatically. The **main( )** may call other functions and some of them might call other functions.

Each function has a unique name and when the name is encountered while in execution the control of the program is transferred to the statement(s) within the function. When the function is ended (returns) the control is resumed to the next statement following the function call.

Well designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions and then each can be called in the proper order.

7.2 Function Prototypes

Using functions in your program requires that you first declare the function (the prototype). Later you can implement the function. The prototype tells the compiler in advance about some characteristics (name of the function, return type and parameters) of a function. A function prototype takes the form:

```
type function-name(type argement-1, type argument-2, .....);
```

The function prototype is a statement that ends with a semicolon. The function name is any legal identifier followed by the function parenthesis without any space in between. The function "type" is the data type of the return value. If the function does not return a value, the data type is defined as **void**. The arguments (or parameters) come inside the parenthesis, preceded by their types and separated by the commas. If the function does not any parameters, it is either kept blank or the word **void** is used inside the parenthesises. Following are some examples for prototypes:

```
void exit(int x);
int kbhit( );
double power(double x, double y);
```

The first function returns no value and it takes an integer argument. Second prototype returns an integer value and has no inputs. Third function returns a double value and takes two double values.

7.3 Function Definition

The definition of a function is the actual body of the function. It starts with function header, which is the same as the function prototype but does not end with a semicolon. The function prototype and the definition must have exactly the same return type, name and parameter list. If they do not match the compiler will issue error messages during compilation. Function definition takes the form:

```
Return-type function-name(type argement-1, type argument-2, ..... )
{
   Statement(s);
}
```

Consider the following function, which accepts an integer as the input and returns its square.

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

Whenever you want to call this function you can include a statement like the following in the program.

```
b = square(a);
```

Here, the value of variable "**a**" is passed to the function, where it is received by the function as the value of variable "**x**". The function returns the square as the value of "**y**". The variable "**b**" in the calling function receives the return value.

Example 7.1 – Write a C program to calculate the circumference and area of a circle given its radius. Implement calculation of circumference and areas as separate functions.

```
/*Program-7.1     */
#include <stdio.h>
const float pi = 3.141;

float area(float r);          //function prototype
float circumference(float r); //function prototype

int main()
{
   float radius;
   printf("Enter radius: ");
   scanf("%f",&radius);       //read radius
   printf("\nArea : %.2f", area(radius));
   printf("\nCircumference : %.2f", circumference(radius));
   return 0;
}

/* Function computes the area of a circle given its radius*/
float area(float r)
{
     return (pi*r*r);
}

/* Function computes the circumference of a circle given radius*/
float circumference(float r)
{
     return (2*pi*r);
}
```

It is also possible to define several functions with the same name but with different parameters. As given below:

```
float area(int r);
float area(float r);
```

Both these functions calculate the area of a circle given the radius. However the first function accepts an integer as the input while the second function accepts a floating point number. Based on the type of the given input the program dynamically calls the correct function. Consider the following program:

```
/*Program-7.2     */
#include <stdio.h>
const float pi = 3.141;       //define pi

void area(float r);           // function prototype
void area(int r);             // function prototype

int main()
{
   int radius;
   printf("Enter radius: ");
   scanf("%d",&radius);

   area(radius);
   area((float)radius);       //convert to float

   return 0;
}

void area(float r)
{
   printf("\nFloating point input");
   printf("\nArea is: %.2f", (pi*r*r));
}

void area(int r)
{
   printf("\nInteger input");
   printf("\nArea is: %.2f", (pi*r*r));
}
```

Execution of Program-7.2 displays:

```
Enter radius: 1

Integer input
Area is: 3.14
Floating point input
Area is: 3.14
```

Program-7.2 executes each function based on the given input. The program accepts an integer from the keyboard and when the expression **area(radius);** is executed it calls the function which accepts and integer as the input parameter and it display the message "**Integer input**" and the area. The expression **area((float)radius);** converts the integer value to a floating point number (this process is called *casting*) before calling the function. When it is called since the input is a floating point value it executes the function which displays the message "**Floating point input**".

Although you can have multiple functions with the same name but with different input parameters, you cannot have functions with the same name with a different output parameter. It is also possible to have the same name for functions with different behaviour with different input parameters, however best practices suggest that no two functions should have the same name unless they perform identical tasks.

7.4 Passing Variables to Functions

You can use as many functions as you need in your programs and you can call a function from another or even from it-self. In order to avoid errors when using functions, you have to have a clear understanding about the mechanism of passing variables from one function to another.

7.4.1 The Scope of a Variable

A variable has a *scope*, which determines how long it is available to your program (or function) and where it can be accessed from. Variables declared within a block are scoped only to that block; they can be accessed only within that block and go out of existence when the execution of the block is completed.

A variable declared inside a function is called a *local variable*. Scope of a local variable is limited to the function. Such variables are not seen by any other functions including the **main** function. When you pass a variable to a function (such as the variable **radius** in Program-7.1), you are actually passing a copy of the variable (called *passed by value*), but not the variable it self. This means that the value of the passed variable cannot be changed by any other function. Even if you use another variable with the same name in the function, you still have two local variables isolated from each other.

On the other hand you can also define *global variables* which are accessible from any function within the same source file. The global variable can be defined within the program but anywhere outside function block including the **main** function. For example you can define a global variable "**discount**" as follows:

```
#include <stdio.h>
float discount;   //global variable
float sub_total(float total);  //function prototype
int main()
{
....
```

A variable defined in such a manner is visible to the **main** function and the "**sub_total**" function. You can modify its value in either place and read it from another place.

A well written C source code should not have any global variables unless they are specifically required since an accidental change to a global variable may produce erroneous results.

7.4.2 Default Parameters

For every parameter you declare in a function prototype and declaration the calling function must pass in a value. The value passed must be of the declared type. Thus if you have a function declared as:

```
long my_function(int a);
```

the function must in fact take an integer value as an input. If the function definition differs or if you pass a value of a wrong data type you will get a compilation error. However when you declare the function prototype you can define a default value for the parameter as follows:

```
long my_function(int a = 50);
```

The default value is used if no input is supplied to the function. Similarly a function with many input parameters can have a default value for each parameter.

Exercise 7.1 – Predict the output of each `printf` function in Program-7.3.

```
/*Program-7.3     */
#include <stdio.h>

void swap1(int x,int y);
void swap2(int x);
int a;

int main()
{
   int b,c;
   a=5;
   b=10;
   c=15;
   printf("\nValue before 1st function a= %d, b= %d c= %d" ,a,b,c);
   swap1(b,c);
   printf("\nValue after 1st function a= %d, b= %d c= %d" ,a,b,c);
   swap2(b);
   printf("\nValue after 2nd function a= %d, b= %d c= %d" ,a,b,c);
   printf("Test");
   return 0;
}

void swap1(int x, int y)
{
   int z;
   z = x;
   x = y;
   y = z;
}

void swap2(int x)
{
   int z;
   z = x;
   x = a;
   a = z;
}
```

8 – Pointers

Pointers are another important feature of the C language. Although it may appear a little confusing for a novice programmer they are a powerful tool and handy to use once they are mastered. The power of C compared to most other languages lies with proper use of pointers. Pointers are useful due to following reasons:

- They enable us to access a variable that is defined outside a function.
- Pointers are more efficient in handling data tables and sometimes even arrays.
- Pointers tend to reduce the length and complexity of a program.
- They increase the execution speed.
- Use of pointers allows easy access to character strings.

Computers use memory to store both instructions and values of variables of a program. The computer's memory is a sequential collection of storage cells with the capacity of a single byte. Each of these memory cells has an address associated with it.

Whenever a variable is declared the system allocates some memory to hold the value of the variable. Such memory location can be accessed by providing the memory address. Consider the following example:

```
int number = 35;
```

The above expression allocates a memory location to hold the value of variable "**number**" that can hold an integer (4 bytes) and it also initialises the variable. Suppose the address of that memory location is 2000. Then after executing above expression the memory address 2000 should hold 35. During execution of the program, the system always associates the name "**number**" with the memory address 2000. We may have access to the value "**35**" by using either the name "**number**" or the address 2000. Since memory addresses are simple numbers, they can also be assigned to some variables. Such variables that hold memory addresses are called *pointers*. Therefore a pointer is nothing but a variable that contains an address which is a location of another variable in memory.

8.1 Declaring Pointers

A pointer is declared using the indirection (*****) operator. The typical declaration of a pointer is:

data-type * pointer-name;

If a pointer "**a**" is pointing to an integer, it is declared as:

```
int *a;
```

Since a pointer is a variable, its value is also stored in another memory location. Therefore in computation even the address of the pointer can be used.

The location of a variable in memory is system dependent and therefore the address of a variable is not known directly. The address operator (**&**) allow us to retrieve the address from a variable associated with it. Consider the following example:

```
/* Program-8.1    */
#include <stdio.h>

int main()
{
   int number = 20;
   int *pnt;
   pnt = &number;

   printf("\nThe number is: %d", number);
   printf("\nThe address of the number is: %d", &number);
   printf("\nThe pointer is: %d", pnt);
   printf("\nThe address of the pointer is: %d", &pnt);
   printf("\nThe value of the pointer is: %d", *pnt);
   return 0;
}
```

Execution of Program-8.1 displays:

```
The number is: 20
The address of the number is: 1245064
The pointer is: 1245064
The address of the pointer is: 1245060
The value of the pointer is: 20
```

The first **printf** function displays the value of variable "**number**". The second **printf** statement displays the address of the memory location occupied by the variable named "**number**". The third statement displays the value of the "**pnt**" which is assigned by the expression **pnt = &number;**. Note that now the address of variable "**number**" and value of pointer "**pnt**" is the same. The fourth **printf** function displays the address of the pointer. The final statement displays the value of the pointer "**pnt**" which holds the value of the variable "**number**".

Example 8.1 – Write a program to swap two integer numbers using pointers.

```
/* Program-8.2    */
#include <stdio.h>
void swap(int *a,int *b);

int main()
{
   int a,b;
   a = 5;
   b = 10;

   printf("\nBefore swapping a= %d: b= %d", a, b);
   swap(&a, &b);                                //call function
   printf("\nAfter swapping a= %d: b= %d", a, b);

   return 0;
}

void swap(int *a, int *b)
{
   int x;
   x = *b;
   *b = *a;
   *a = x;
}
```

Execution of Program-8.2 displays:

```
Before swapping a= 5: b= 10
After swapping a= 10: b= 5
```

8.2 Text Strings and Pointers

An array of characters is called a *string*. Strings in C are handled differently than most other languages. A pointer is used to keep track of a text string stored in memory. It will point to the first character of the string. By knowing the beginning address and the length of the string, the program can locate it.

A character pointer is used to point to the first character of a string as given in the following example:

```
char *a;
a = "Hello World!";
```

Consider the following example:

```
/* Program-8.3    */
#include <stdio.h>

int main()
{
   char *a;
   a = "Hello World";

   printf("String: %s\n", a);
   printf("First character: %c\n", *a);
   printf("Starting memory address: %d\n", a);
   printf("First character: %d\n", *a);

   return 0;
}
```

Exaction of Program-8.3 will display:

```
String: Hello World
First character: H
Starting memory address: 4235496
First character: 72
```

In Program-8.3 the first **printf** function displays the string pointed by pointer "**a**". The second **printf** function display the value pointed by pointer "**a**" which is the first character of the string. The third **printf** function displays the starting memory address of the string which is the value of pointer "**a**". The final **printf** function displays the ASCII value of the first character in the string.

8.3 Pointers and Arrays

An array is a series of elements of the same data type. Pointers can be used to manipulate arrays rather than using an index. The name of an array points to the first element of the array. If you declare an array in the following manner:

```
int marks[5];
```

you can point to the first element of the array using either one of the following pointers:

```
marks            //first element
&marks[0]        //first element
```

Also the following pairs of pointers are equivalent:

```
marks+1 == &marks[1]
......
marks+4 == &marks[4]
```

Or you can use the array name to refer to the contents of the array elements like:

```
*(marks)         //value of 1st element
*(marks+1)       //value of 2nd element
```

Program 8.4 illustrates the use of array name, index and pointers.

```
/* Program-8.4    */
#include <stdio.h>

int main()
{
   int marks[5]= {89, 45, 73, 98, 39};

   printf("%d\n", marks);       //memory address pointed by pointer
   printf("%d\n", &marks[0]);   //memory address of 1st element
   printf("%d\n", *marks);      //value pointed by pointer
   printf("%d\n", marks[0]);    //value of 1st array element
   return 0;
}
```

9 – Handling Files

The programs that we developed up to now were neither able to produce permanent output nor were they able to read data inputs other than from the keyboard. Using files you can save your output data permanently and retrieve them later.

A file in general is a collection of related *records*, such as student information, marks obtained in an exam, employee salaries, etc. Each record is a collection of related items called *fields*, such as "student name", "date of birth", "subjects registered", etc. The common operations associated with a file are:

- Read from a file (input)
- Write to a file (output)
- Append to a file (write to the end of a file)
- Update a file (modifying any location within the file)

In C language data is transferred to and from a file in three ways:

- Record input/output (one record at a time)
- String input/output (one string at a time)
- Character input/output (one character at a time)

9.1 The File Protocol

Accessing a file is a three step process:

- Opening a connection to a file
- Reading/writing data from/to the file
- Closing the connection

9.1.1 Opening a Connection to a File

In order to use a file on a disk you must establish a connection with it. A connection can be established using the **fopen** function. The function takes the general form:

```
fopen(file_name, access_mode)
```

The **file_name** is the name of the file to be accessed and it may also include the path. The **access_mode** defines whether the file is open for reading, writing or appending data. Table 9.1 summarises the access modes supported by **fopen** function.

Table 9.1 – File access modes

Access mode	**Description**
"r"	Open an existing file for reading only.
"w"	Open a file for writing only. If the file does not exist create a new one. If the file exists it will be overwritten.
"a"	Open a file for appending only. If the file does not exist create a new one. New data will be added to the end of the file.
"r+"	Open an existing file for reading and writing
"w+"	Open a new file for reading and writing
"a+"	Open a file for reading and appending. If the file does not exist create a new one.

The following code opens a file named "my file.txt" in the current directory for appending data:

```
FILE *fp;
fp = fopen("my file.txt", "a");
```

The function **fopen** returns a pointer (referred as the *file pointer*) to the structure[6] **FILE** which is defined in the **stdio.h** headier file. When you open a file it would be better to make sure that the

[6] Structures are used to store records with different data types.

operation is successful. If the establishment of a connection is successful the function returns a pointer to the file. If an error is encountered while establishing a connection the functions returns **NULL**.

9.1.2 Closing the Connection to a File

After a connection to a file is established it can be used to read or write data. When all the file processing is over the connection should be closed. Closing the connection is important as it writes any remaining data in the buffer to the output file. The function **fclose** is used to close the file. For example:

```
fclose(fp)
```

When closing the file the file pointer "**fp**" is used as an argument of the function. When a file is successfully closed the function **fclose** returns a **zero** and any other value indicates an error.

9.1.3 Reading Data from a File

When reading data from a ASCII file you can either read one character or one string at a time.

Reading Characters from a File

To read one character at a time you can use the **getc** function. It takes the form:

```
getc(file_pointer)
```

You can assign the output of the function **getc** to an **int** or **char** variable.

Example 9.1 – Write a program to read the file "my file.txt" which has the message:

```
Hello World!
This is my first file
```

The following program reads the file "my file.txt" one character at a time and displays it on the screen.

```
/* Program-9.1    */
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("my text.txt", "r"); //open read-only
   if(fp != NULL)
   {
      c = getc(fp);            //read the 1st character
      while ( c != EOF)        //if not the end of file
      {
         printf("%c",c);
         c= getc(fp);          //read next character
      }
      fclose(fp);              //close the file
   }
   else
      printf("\nError while opening file...");

   return 0;
}
```

Execution of Program-9.1 will display

```
Hello World!
This is my first file
```

In Program-9.1 a connection is first established to the file. Then the expression **if(fp != NULL)** evaluates whether the connection is successful. If it is not successful the program will display the message "**Error while opening file...**" If it is successful it reads the first character from the file. If the character is not the end of the file (indicated by the End Of File (**EOF)** mark) it displays the character. Then the program continues to read the rest of the characters in the file until it finds the **EOF** mark. Afterwards the connection to the file is closed using the **fclose** function.

Reading a String from a File

In real-life applications it is more useful to read one string at a time rather than one character. With every read, the program has to check for the line feed (LF) character so it can find the end of each string. Also it must check for the EOF mark which comes at the end of the file. The **fgets** function can be used to read a string at a time. The function generally takes the form:

fgets(string, max_characters, file_pointer)

The "**string**" is a character array (also called a character *buffer*) and "**max_characters**" define the maximum number of characters to read form a line. The function **fgets** returns a **char** pointer. It returns **NULL** if **EOF** mark is encountered. One deficiency in **fgets** is that it can only read to a fixed character buffer, therefore you need to know in advance the maximum number of characters in a string.

Example 9.2 – Modify Program-9.1 such that it uses the fgets function instead of fgetc function.

Suppose the file does not have more than 100 characters in a line.

```
/* Program-9.2    */
#include <stdio.h>

int main()
{
   FILE *fp;
   char buffer[100];    //char array with 100 elements
   char *result;        // hold the result of the fgets function

   fp = fopen("my text.txt", "r"); //open read-only
   if(fp != NULL)
   {
      result = fgets(buffer, 100, fp); //read the 1st string
      while(result != NULL)            //if not the end of file
      {
         printf("%s",buffer);
         result = fgets(buffer, 100, fp); //read the next string
      }
      fclose(fp);                         //close the file
   }
   else
      printf("\nError while opening file");

   return 0;
}
```

9.1.4 Writing Data to a File

You can also write data to file either one character at a time or a string at a time.

Writing Character to a File

To write a character to a file the **putc** function can be used. It has the form:

putc(c, fp)

where **c** is the character while **fp** is the file pointer. It returns an **int** value which indicates the success or the failure of the function. It returns the **int** value of the character if it is successful, if not it returns **EOF** mark.

Example 9.3 – Write a C program to store the message "Introduction C Programming" in a file named "message.txt".

Program-9.3 is an implementation of the above requirement. The function **putc** is used to write characters in the message to the file. To find the number of characters in the message the **strlen** function which returns the number of characters in a string is used. A pointer is used to point to the string and the pointer is incremented by one memory location at a time.

```
/* Program-9.3    */
#include <stdio.h>
#include <string.h>

int main()
{
   FILE *fp;
   int i;               //loop counter
   char *message;
   message = "Introduction C Programming";

   fp = fopen("c:\\message.txt", "w"); //open for writing
   if(fp != NULL)                      //if success
   {
      for (i =0 ; i < strlen(message);i++)
         putc(*(message+i),fp); //write character pointed by pointer
      fclose(fp);               //close the file
   }
   else
      printf("\nError while opening file");

   return 0;
}
```

Writing a String to a File

The advantage of **putc** is that it allows you to control every byte that you write into the file. However sometimes you may want to write one string at a time. Two functions, namely **fputs** and **fprintf** can be used for this purpose. The **fprintf** function is identical to the **printf** function only difference being that it writes to a file rather than to the screen. The format of each function is:

fputs(string, file_pointer)
fprintf(file_pointer, “% s”, string)

Exercise 9.1 – Modify Program-9.3 such that it uses the fputs rather than the fputc function to write the message to the file.

Exercise 9.2 – Develop a simple telephone directory which saves your friends contact information in a file named directory.txt. The program should have a menu similar to the following:

```
----------------Menu-------------------------
1. Add new friend.
2. Display contact info.
3. Exit
------------------------------------------------
Enter menu number:
```

When you press “1” it should request you to enter following data:

```
---------New friend info--------
Name : Saman
Phone-No: 011-2123456
e-Mail : saman@cse.mrt.ac.lk
```

After adding new contact information it should again display the menu. When you press “2” it should display all the contact information stored in the directory.txt file as follows:

```
-------------Contact info---------------
Name        Tel-No           e-Mail
Kamala      077-7123123      kamala@yahoo.com
Kalani      033-4100101      kalani@gmail.com
Saman       011-2123456      saman@cse.mrt.ac.lk
-----------------------------------------
```