2. STRUCTURE OF THE C PROGRAM

Every C program consists of one or more modules called functions. One of the functions must be called *main*. The program will always begin by executing the main function. Any other function definition must be defined separately, either ahead or after main. Each function must contain:

1. A function heading, which consists of the function name, followed by an optional list of arguments, enclosed in parentheses.
2. A list of argument declarations, if arguments are included in the heading.
3. A compound statement, which comprises the remainder of the function.

Each compound statement is enclosed within a pair of braces, i.e., {}. These braces may contain one or more elementary statements (called expression statements) and other compound statements. Each expression statement must end with a semicolon (;).

Comments (remarks) may appear anywhere within a program, as long as they are placed in within the delimiters, /* and */. Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

Here is an elementary C program, which illustrates the overall program organization.

```
/* Program to calculate the area of a rectangle */
#include<stdio.h>                          /* Library file name */
main()                                     /* Function heading */
{   float length,breadth,area;             /* Variable declaration */
    clrscr();                              /* Library function call */
    printf("Enter the length : ");         /* Output prompt message */
    scanf("%f",&length);                   /* Input statement */
    printf("Enter the breadth : ");        /* Output prompt message */
    scanf("%f",&breadth);                  /* Input statement */
    area = length*breadth;                 /* Assignment statement */
    printf("Area = %f",area);              /* Output statement */
}
```

3. C PROGRAM DEVELOPMENT ENVIRONMENT

Developing a program in a compiled language such as C requires the following steps:

1. Editing the program;
2. Pre-processing
3. Compiling the program;
4. Linking the program with functions that are needed from the C library; and

One can also use any one of the several cross-platform IDEs available for developing software using C. Some of these IDEs are:

NETBEANS

Netbeans is a free, open-source and popular cross-platform IDE for C/C++ and many other programming languages. Most people known Netbeans as a Java IDE. Actually, you can also use this tool to develop a C/C++ based application. Its fully extensible using community developed plugins. All standard features of a C/C++ IDE are fulfilled by NetBeans IDE, including code-completion and debugger.

The C/C++ editor of NetBeans IDE is well integrated with the multi-session GNU gdb debugger. You can set variable, exception, system call, line, and function breakpoints and view them in the Breakpoints window.

You can also use the development tools of this IDE on remote hosts to build, run, and even debug projects from your client system as simple as if it is done locally. NetBeans IDE is free to use. It is also released as an open source software.

ECLIPSE

Eclipse is mostly written in Java and it is primarily used for developing Java applications. But, the language support can be extended by installing plugins. So with plugins support, Eclipse becomes one of the best IDEs to develop programs in C, C++, COBOL, Fortran, Haskell, JavaScript, PHP, Perl, Python, R, Ruby and Ruby on Rails, Scheme and many more. Eclipse SDK (Software Development Kit) is free and open source.

CODE::BLOCKS

Code::Blocks is a free and open source, cross-platform IDE which supports multiple compilers including GCC. Code::Blocks is oriented towards C, C++ and Fortran. It delivers a consistent user interface and feel.

And most importantly, you can extend its functionality by using plugins developed by users, some of the plugins are part of Code::Blocks release and many are not, written by individual users not part of the Code::Block development team. Its features are categorized into compiler, debugger and interface features

CODELITE

CodeLite (Figure 1.4) is another free IDE you can use to develop a C/C++ based application. This tool is also released as an open source software. In addition to C/C++, this IDE also supports such programming languages as PHP and JavaScript. If you use it to develop a C/C++ application, you can get the features like code completion. You don't also have to manually compile your code on the terminal since CodeLite also features compilers.

ATOM CODE EDITOR

Atom is completely hackable, which means you can customize it as you want. It supports large number of programming languages by default like php, javascript, HTML, CSS, Sass, Less, Python, C, C++, Coffeescript, etc. and if you working with a language that is not supported by default in Atom, you can install plugin for the purpose.

BLUEFISH EDITOR

Bluefish is a more than just a normal editor, it is a lightweight, fast editor that offers programmers IDE like features for developing websites, writing scripts and software code. It is multi-platform, runs on Linux, Mac OSX, FreeBSD, OpenBSD, Solaris and Windows, and also supports many programming languages including C/C++.

BRACKETS CODE EDITOR

Brackets is a modern and open-source text editor designed specifically for web designing and development. It is highly extensible through plugins, therefore C/C++ programmers can use it by installing the C/C++/Objective-C pack extension, this pack is designed to enhance C/C++ code writing and to offer IDE like features.

SUBLIME TEXT EDITOR

Sublime Text is a well refined, multi-platform text editor designed and developed for code, markup and prose. You can use it for writing C/C++ code and offers a great user interface.

KDEVELOP

KDevelop is just another free, open-source and cross-platform IDE that works on Linux, Solaris, FreeBSD, Windows, Mac OSX and other Unix-like operating systems. It is based on the KDevPlatform, KDE and Qt libraries. KDevelop is highly extensible through plugins.

GEANY IDE

Geany is a free, fast, lightweight and cross-platform IDE developed to work with few dependencies and also operate independently from popular Linux desktops such as GNOME and KDE.

AJUNTA DEVESTUDIO

Ajunta DevStudio is a simple GNOME yet powerful software development studio that supports several programming languages including C/C++. It offers advanced programming tools such as project management, GUI designer, interactive debugger, application wizard, source editor, version control plus so many other facilities.

THE GNAT PROGRAMMING STUDIO

The GNAT Programming Studio is a free easy to use IDE designed and developed to unify the interaction between a developer and his/her code and software. Built for ideal programming by facilitating source navigation while highlighting important sections and ideas of a program. It is also designed to offer a high-level of programming comfortability, enabling users to developed comprehensive systems from the ground.

4. THE C CHARACTER SET

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form to basic program elements (e.g., constants, variables, operators, expressions, etc.). The special characters are listed below.

+ - * / = & # ! ? ^ " ' ~ \ |
< > ( ) [ ] { } : ; . _ % Blank Space

C uses certain combinations of these characters, such as \b, \n, \t, <=, >=, &&, ||, !=, etc., to represent special conditions.

5. DATA TYPES

The C language defines five fundamental data types: character, integer, floating-point, double floating-point and valueless. These are declared using the keywords *char, int, float, double* and *void*, respectively. These types form the basis for several other types. The size and the range of these data types may vary among processor types and compilers. Typical memory requirements of the basic data types are given below.

Data type	Meaning	Size (byte)	Minimal Range
char	Character	1	-128 to 127
int	Integer	2	-32,768 to 32767
float	Single precision real number	4	3.4E-38 to 3.4E+38 with 6 digits of precision
double	Double precision real number	8	1.7E-308 to 1.7E+308 with 10 digits of precision
void	Valueless	0	

void is used to specify an empty set of values

5.1. DATA TYPE QUALIFIERS (TYPE MODIFIERS)

Except type *void*, the basic data types may have various *modifiers* or *qualifiers* preceding them. A data type qualifier alters the meaning of the base type to more precisely fit a specific need. The commonly used data type qualifiers are *signed, unsigned, short* and *long*. The *int* base type can be modified by *signed, unsigned, short* and *long*. The *char* can be modified by *signed* and *unsigned*. One may also apply *long* to *double*.

The interpretation of qualified integer data type will vary from one C compiler to another, though there are some common sense relationships. Thus, a *short int* may require less memory than an ordinary *int* or it may require the same amount of memory as an ordinary *int*, but it will never exceed an ordinary *int* in word length. Similarly, a *long int* may require the same amount of memory as an ordinary *int* or it may require more memory, but it will never be less than the ordinary *int*.

If *short int* and *int* both have the same memory requirements (e.g., 2 bytes) then *long int* will generally have double the requirements (e.g., 4 bytes) or if *int* and *long int* both have the same memory requirements (e.g., 4 bytes) then *short int* will generally have half the memory requirements (e.g., 2 bytes).

The use of *signed* on integers is allowed, but it is redundant because the default integer declaration assumes a signed number. The most important use of *signed* is to modify *char* in implementations in which *char* is unsigned by default.

An *unsigned int* has the same memory requirement as an ordinary *int*. However, in the case of an ordinary *int* (or a *short int* or *long int*), the left most bit is reserved for the sign. With an *unsigned int*, all of the bits are used to represent the numerical value. Thus, a *signed int* can be approximately twice as large as an ordinary *int* (though, of course, negative values are not permitted). For example, if an ordinary *int* can vary form –32,768 to +32767 (which is typical for 2-byte *int*), then an *unsigned int* will be allowed to vary from 0 to 65,535. The *unsigned* qualifier can also be applied to other qualified integers, e.g., *unsigned short int* or *unsigned long int*.

Some compilers permit the qualifier *long* to be applied to *float* or *double*, e.g., *long float*, or *long double*. However, the meaning of these data types will vary from one C compiler to another. Thus *long float* may be equivalent to *double*. Moreover, *long double* may be equivalent to *double*, or it may refer to a separate extra-large double precision data type requiring more than 8 bytes of memory.

When a type quantifier is used by itself (i.e., when it does not precede a basic type), then *int* is assumed. For example, the type specifies *signed, unsigned, short* and *long* are same as *signed int, unsigned int, short int* and *long int*, respectively.

The following table shows all valid data type combinations supported by C, along with their minimal ranges and typical memory size.

Type	Memory Size (bytes)	Minimal Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to +127
int	2 or 4	-32,768 to 32,767
unsigned int	2 or 4	0 to 65,535
signed int	2 or 4	Same as int
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
signed short int	2	Same as short int
long int	4	-2,147,483,647 to 2,147,483,647
signed long int	4	Same as long int
unsigned long int	4	0 to 4,294,967,295
float	4	1E-38 to 1E+38 with 6 digits of precision
double	8	1E-308 to 1E+308 with 10 digits of precision
long double	8	1E-308 to 1E+308 with 10 digits of precision

6. TOKENS

The smallest individual elements, which are identified by the compiler, are known as tokens. Tokens supported in C can be categorized as:

- Identifiers
- Keywords
- Constants
- Variables
- Operators

6.1. IDENTIFIERS

Identifiers are the names that are given to various program elements, such as variables, functions and arrays. The rules of forming an identifier are:

1. Identifier must begin with a letter of alphabet. In C, the underscore (_) character is considered a letter.
2. The first character may be followed by a sequence of letters and/or digits (0 through 9).
3. An identifier can be arbitrarily long. However some implementations of C recognize only the first 8 characters.
4. Both upper- and lower-case characters are permitted. However, they are not interchangeable, i.e., an uppercase letter is not equivalent to the corresponding lowercase letter.
5. No identifier may be a keyword.
6. No special characters, such as blank space, period, semicolon, comma, or slash, are permitted.
7. An identifier should contain enough characters so that its meaning is readily apparent. On the other hand, an excessive number of characters should be avoided.

The following names are valid identifiers.

Count total sum_1 temperature tax_rate TABLE

The following names are not valid identifiers for the reason stated.

4th	The first character must be a letter of alphabet
order-no	Illegal character (-)
error flag	Illegal character (blank space)
int	Keyword

6.2. KEYWORDS

Keywords are the standard identifiers that have standard, predefined meaning in C. These keywords can be used only for their intended purpose and they cannot be used as programmer-defined identifiers. The standard keywords are:

auto break case char

const continue default do

double else enum extern

float for goto if

int long register return

short signed sizeof static

stuct switch typedef union

unsigned void volatile while

Note that keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier, but it is not a good programming practice.

6.3. CONSTANTS OR LITERALS

The term constant in C refers to fixed values that does not change during the execution of a program. There are four basic types of constants in C. They are:

1. Integer constants
2. Floating-point constants
3. Character constants
4. String constants

Integer and floating-point constants represent numbers. They are often referred to collectively as numeric-type constants. The following rule applies to all numeric type constants:

- Comma and blank spaces cannot be included within the constants.
- Constants can be preceded by a – or + sign, if desired. If either sign does not precede the constant it is assumed to be positive.
- The value of a constant cannot exceed specified minimum and maximum bounds. For each type of constant, these bounds vary from one C compiler to another.

INTEGER CONSTANTS

Integer constants are whole numbers without any fractional part. Thus integer constants consist of a sequence of digits. Integer constants can be written in three different number systems: Decimal, Octal and Hexadecimal.

A decimal integer constant can consist of any combination of digits taken from the set 0 through 9. If the decimal constant contains two or more digits, the first digit must be something other than 0.

The following are valid decimal integer constants

0 1 1234 743 -8321

The following decimal integer constants are written incorrectly for the reasons stated.

12,345	Illegal character (,).
132.0	Illegal decimal point (.).
10 20 30	Illegal character (blank space).
098	First digit cannot be zero.

An octal integer constant can consists of any combination of digits taken from the set 0 through 7. However, the first digit must be 0, in order to identify the constant as an octal number.

The following are valid octal integer constants.

0 01 0743 -0743 +04232

The following octal integer constants are written incorrectly for the reasons stated.

743	Does not begin with 0
05283	Illegal digit (8)
07.32	Illegal character (.)

A hexadecimal integer constant must begin with either 0x or 0X. It can then be followed by any combination of digits taken from the sets 0 through 9 and A through F (either upper- or lower-case).

The following are valid hexadecimal integer constants

0X0 0x1 0X7FAB 0xabcd -0xface

The following hexadecimal integer constants are written incorrectly for the reasons stated.

0x12.34	Illegal character (.)
0ABCDE	Does not begin with ox.
oxagfedc	Illegal character (g)

QUALIFIED INTEGER CONSTANTS

The data type qualifier *short, long, signed* and *unsigned* may be applied to any integer. Unsigned integers are positive. It can be used to increase the range of positive numbers normally stored. Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of 2, though they may not be negative. An unsigned integer constant can be identified by appending the letter *U* (either upper- or lower case) to the end of the constant.

Long integer constant may exceed the magnitude of ordinary integer constants, but require more memory within the computer. A long integer constant can be identified by appending *L* (either upper- or lower case) to the end of the constant. An unsigned long integer may be specified by appending *UL* (either upper- or lower case) to the end of the constant.

Examples:

Data	Description
2468	Decimal integer
-246878325L	Decimal long integer
2468U	Decimal unsigned integer
24687835UL	Decimal unsigned long integer
0123456L	Octal long integer
07777U	Octal unsigned integer
01234567UL	Octal unsigned long integer
0X5000U	Hexadecimal unsigned integer
0XFFFFFUL	Hexadecimal unsigned long integer

FLOATING-POINT CONSTANTS

A floating-point constant is a base-10 number that contains either a decimal point or an exponent or both. A floating-point constant can be written in two forms: Fractional form or Exponential form. A floating-point constant in a fractional form must have at least one digit each to the left and right of the decimal point. A floating-point in exponent form consists of a mantissa and an exponent. The mantissa itself is represented as a decimal integer constant or a decimal floating-point constant in fractional form. The mantissa is followed by the letter E or e and the exponent. The exponent must be a decimal integer. The actual number of digits in the mantissa and the exponent depends on the computer being used.

The following are valid floating-point constants

1.0	0.2	872.602	5000.0
0.000743	315.0066	2E-8	0.006e-3
1.666E+8	.121212e12	-0.156e-4	0.01540e05

The following are some invalid floating-point constants.

1	No decimal point or exponent
1,00.0	Illegal character (,)
2E+10.2	Exponent must be an integer
3 E10	Illegal character (space)

CHARACTER CONSTANTS

A character constant is a single character, enclosed in single quotation marks.

e.g., 'A' 'X' '3' ' '

Characters are stored internally in computer as coded set of binary digits, which have positive decimal integer equivalents. The value of a character constant is the numeric value of the character in the machine's character set. This means that the value of a character constant can vary from one machine to the next, depending on the character set being used on the particular machine. For example, on ASCII machine the value of '*A*' is 65 and on EBCDIC machine it is 193.

ESCAPE SEQUENCES

Certain non-printing characters, as well as backslash (\) and apostrophe ('), can be expressed in terms of escape sequences. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, the newline character (line feed) can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters. The commonly used escape sequences are:

Character	Escape Sequence
Bell (Alert)	\a
Backspace	\b
Horizontal tab	\t
Vertical Tab	\v
Newline	\n
Carriage return	\r
Form feed	\f
Quotation mark (")	\"

Character	Escape Sequence
Apostrophe (')	\'
Question mark (?)	\?
Backslash	\\
Null	\0

The following are some character constants, expressed in terms of escape sequences.

'\n' '\t' '\b' '\"' '\\' '\0' '\''

Of particular interest is the escape sequence \0. This represents the null character (ASCII 000), which is used to indicate the end of string. Note that the character constant '0' is different from '\0'

STRING CONSTANTS

A string constant consists of a set of zero or more characters enclosed in quotation marks. Several string constants are given below.

"green"	"Welcome to C programming"	""
" "	"The largest number is"	"2*(i+3)/j"
" "	"Line no.1\nLine no.2\nLine no.3"	"Error\a\a\a"

Note that the string constant "Line no.1\nLine no.2\nLine no.3" extends over three lines, because of the newline characters that are embedded within the string. Thus, the string would be displayed as

```
Line no.1
Line no.2
Line no.3
```

Sometimes some special characters (e.g., backslash or quotation mark) must be included as a part a string constant. These characters must be represented in terms of their escape sequences. Similarly, certain nonprinting characters (e.g., tab, newline) can be included in a string constant if they are represented in terms of their corresponding escape sequences.

The following string constant includes three special characters that are represented by their corresponding escape sequences.

"\t To continue, press the \"Return\" key ...\n"

The special characters are \t (horizontal tab), \" (double quotation marks, which appears twice), and \n (newline).

The compiler automatically places a null character (\0) at the end of every string constant, as the last character within the string (before closing the double quotation mark). This character is not visible when the string is displayed. However, we can designation eliminates the single character

Remember that a character constant (e.g., ‘*A*’) and the corresponding integer string (“*A*”) are not equivalent. The character constant has an equivalent integer value, whereas a single-character string constant does not have an equivalent the value, and in fact, consist of two characters – the specified character followed by null character (\0).

For example, the character constant ‘*A*’ has an integer value of 64 in ASCII character set. It does not have a null character at the end. In contrast, the string constant “*A*” actually consists of two characters – the upper case letter *A* and the null character \0. This constant does not have a corresponding integer value.

6.4. VARIABLES

A variable is named location in memory that is used to hold a value that can be modified by the program. Thus variables are the identifiers that are used to represent some specified type of information (data items) within designated portions of the program. Each variable has a specified storage location in memory where its data item is stored. The variable is given a name and the variable name is the ‘name tag’ for the storage location. The value of the variable at any instant during the execution of program is equal to the data item stored in the storage location identified by the name of the variable. The data item must be assigned to variable at some point in the program. The data item can be accessed later in the program simply by referring to the variable name.

A given variable can assign different data items at various places within the program. Thus, the information represented by the variable can change during the execution of the program. However, the data type associated with the variable cannot change.

VARIABLES

the right side of an expression like any other variable.

ANSI standard defines another qualifier *volatile* that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). For example:

```
volatile int date;
```

The value of *date* may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as *volatile*, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as *volatile* can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both *const* and *volatile* as shown below:

```
volatile const int location = 100;
```

USER-DEFINED DATA TYPES (*typedef* & *enum*)

C supports the use of user-defined data type for programmer convenience and for improving the logical clarity of the program. Two such user-defined data types are type definition (*typedef*) and enumeration (*enum*).

the above outputs the following results:

```
Initial value of num=2147483647
After incrementing by 1, num=-2147483648
Value of 2147483648+2147483648=0
```

Underflow is the opposite of overflow. While we reach the upper limit in case of overflow, we reach the lower limit in case of underflow. Thus after decrementing 1 from integer variable having a value of -2147483648 (lower bound of the range of values that can be stored in 32 bits), the returned value will be 2147483647. Here we have rolled over from the lowest value of int to the maximum value.

Similar overflow and underflow can occur with all other data types too. In C the overflow and underflow are more serious because there is no warning or exception raised by the C compiler when such a condition occurs. It simply gives incorrect results. We should therefore exercise a greater care to define correct data types for handling the input/ output values.

Some developers argue that the program should either crash or raise exception in such case but the decision for adding such behaviour is in the hands of creators of programming language. By looking at a problem in your program, you can't straightway tell that an overflow or underflow condition has occurred. It is only after debugging that we come to know of the real cause.

6.5. SYMBOLIC CONSTANTS

A symbolic constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant, or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. Symbolic constants are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc., that the symbolic constants represent.

A symbolic constant is defined by writing

```
#define identifier text
```

where *identifier* represents a symbolic name, typically written in upper-case letters and *text* represents the sequence of characters that is associated with the symbolic name. Note that the *text* does not end with a semicolon, since a symbolic constant definition is not a true C statement.

For example, a C program contains the following symbolic constant definitions.

```
#define TAXRATE 0.23
#define PI 3.141593
#define TRUE 1
#define FALSE 0
#define MESSAGE "Welcome"
```

Notice that the symbolic names are written in upper case, to distinguish them from ordinary C identifiers. Also note that the definitions do not end with semicolons.

Now suppose that the program contains the statement

```
area = PI * radius * radius;
```

During the compilation process, each occurrence of symbolic constant will be replaced by its corresponding text. Thus, the above statement will become

```
area = 3.141593 * radius * radius;
```

Now suppose that the semicolon had been incorrectly included in the definition for *PI*, i.e.,

```
#define PI 3.141593;
```

The assignment statement for *area* would then become

```
area = 3.141593; * radius * radius;
```

Note the semicolon preceding the first asterisk. This is clearly incorrect, and causes an error in compilation.

The substitution of the text for a symbolic constant will be carried out anywhere beyond the *#define* statement, except within a string. Thus any text enclosed in double quotation marks will be unaffected by this substitution process.

For example, a C program contains the following statements.

```
#define CONSTANT 6.023
int c;
...
printf("CONSTANT = %f", c);
```

The *printf* statement will be unaffected by symbolic constant definition, since the term "CONSTANT = %f" is string constant. If, however, the *printf* statement were written as

```
printf("CONSTANT = %f", CONSTANT);
```

The *printf* statement would become

```
printf("CONSTANT = %f", 6.023);
```

during the compilation process.

The use of symbolic constant is recommended when writing C programs, since they contribute to the development of clear, orderly programs. For example, symbolic constants are more readily identified than the information they represent, and the symbolic name usually suggests the significance of their associated data items. Furthermore, it is easier to change the value of single symbolic constant than to change every occurrence of some numerical constant that may appear in several places within the program.

The symbolic constants defined using #define comes under the general category of macro instructions (commonly called macros) and it is one of the pre-processor directives used in C language. The pre-processor replaces every occurrénces of the symbolic constant with its actual value. The resulting program no longer includes the directive (since the directives are only for the processor, not the compiler). Remember that all pre-processor directives begin with the sharp sign (#) and they must start in the first column (or at least first nonblank column), and most C compilers there can be no space between # sign and the directive. The directive is terminated not by semicolon, but by the end of line on which it appears. Only one directive can occur on a line.

The following rules apply to a *#define* directive which define a symbolic constant:

1. Naming of the symbolic constants obeys the same rules as variable names. However, by convention, symbolic names are written in CAPITALS to visually distinguish them from the normal variable names.
2. No blank space between the pound sign and the word define is permitted.
3. # must be the first character in the line.
4. A blank space is required between #define and symbolic name and between the symbolic name and the constant.
5. *#define* statements must not end with a semicolon.