CHAPTER

CONSTANTS, VARIABLES AND DATA TYPES

7.1 INTRODUCTION

To communicate with a computer, it is necessary to pass values from the keyboard to the computer. Such values that are passed may be of different types namely, a single character, an integer, or a decimal string which requires memory area of a different size. Such defined value types are termed as data types. The defined type of memory area where the value is stored is termed as variables. The values which do not change during the execution of the program are termed as constants.

These kinds of data consisting of numbers, characters, and strings are processed using a programming language to provide useful output known as *information*. The data is being processed by executing a sequence of precise instructions called a *program.* According to some rigid rules known as *syntax rules,* these instructions are formed using certain symbols and words. Every program instruction must confirm to the syntax rules of the language.

7.2 CHARACTER SET

The character set is the fundamental set of any language. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special Characters
4. Whitespace

The character set of C can be represented as follows:

The words are separated using whitespaces, but spaces are prohibited between the characters of keywords and identifiers.

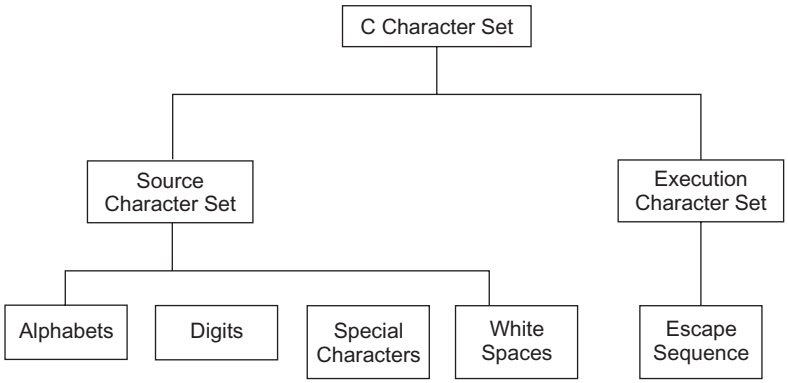

Fig. 7.1 *Character Set of C*

7.2.1 Source Character Set

They are used to construct the statements in the source program.

Example:

Table 7.1 *Source Character Set*

S. No.	Source Character Set	Notation
1.	Alphabets	A to Z and a to z
2.	Decimal Digits	0 to 9
3.	White Spaces	Blank space
4.	Special Characters	+,-,*,/,%,$,# etc.

7.2.2 Execution Character Set

These are employed at the time of execution. This set of characters is also called as non graphic characters because these characters are invisible and cannot be printed directly. These characters will have effect only when the program is executed. They are also called as escape sequences.

Example:

Table 7.2 *Execution Character Set*

S. No.	Escape Sequence	Result
1.	\a	Beep Sound
2.	\t	Moves next horizontal tab
3.	\n	Moves next line
4.	\v	Moves next vertical tab
5.	\0	Null

7.2.3 Trigraph Characters

The concept of trigraph sequences to provide a way to enter certain characters that are not available on some keyboards is introduced in ANSI C. Each trigraph sequence consists of three characters, i.e., two question marks followed by another character. For example, if a keyboard does not support backslash, it can be used in a program using the trigraph **??/.**

Table 7.3 *Trigraph Characters*

Trigraph Sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vertical bar
??/	\ backslash
??-	tilde

7.3 C TOKENS

Individual text and punctuation in a passage of text are usually referred as the tokens. The C program contains individual units called the C tokens. C has the following six types of tokens:

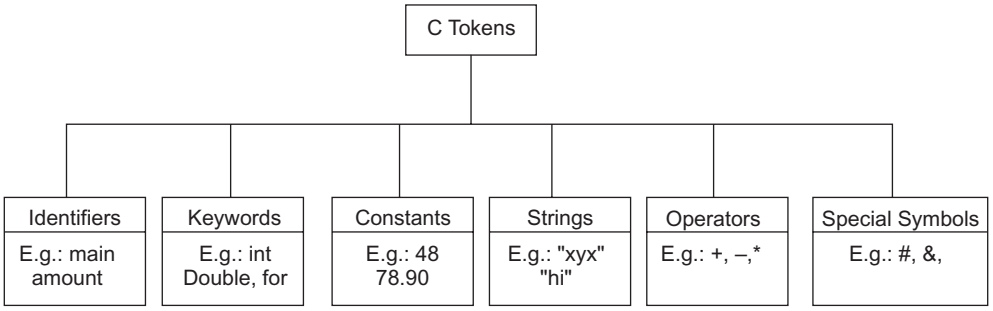

Fig. 7.2 *C Tokens and Examples*

7.4 IDENTIFIERS AND KEYWORDS

In C language, every word is classified into either a keyword or an identifier.

Identifiers: Identifiers are the names given to variables, functions and arrays etc. These are user defined names and consist of a sequence of letters and digits with a letter as the first character. Although

lowercase letters are commonly used, both uppercase and lowercase letters are permitted. The underscore character is also permitted which is used as a link between two words in long identifiers.

Rules for naming an identifier:

- Identifiers consist of letters, digits or underscore in any order.
- The first character must be a letter or character or may begin with underscore.
- An identifier can be of any length while most of the C compiler recognizes only the first 31 characters.
- No white space and special symbols are allowed between the identifier.
- The identifier cannot be a keyword.

Examples for valid Identifiers:

stdname, sub, tot_marks

Examples for invalid Identifiers:

Return, std name, * say

7.5 KEYWORDS

Keywords are reserved words that have standard and predefined meaning in C language. They are the basic building blocks for program statements. Some of the keywords are:

int, break, continue, for, float, char etc.

Table 7.4 *ANSI C Keywords*

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

7.6 CONSTANTS

Constants are data items whose values cannot be changed during the execution of program. They are classified as follows:

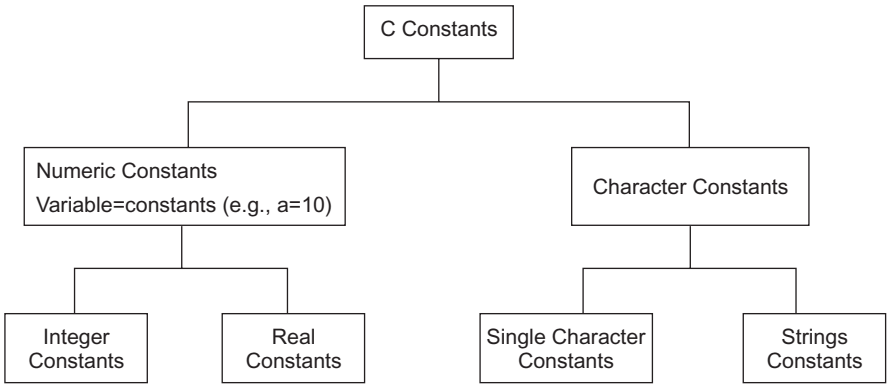

Fig. 7.3 *Types of C Constants*

7.6.1 Numeric Constants

(a) Integer Constants: An integer constant is formed with the sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. For example,

67, 90, 0, +56, –45

Commas, spaces, non digit characters are not permitted between the digits. For example,

10,000 $200, 12 900

are illegal numbers.

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. For example,

067 0 0231

A hexadecimal integer is defined as the sequence of digits preceded by 0x or 0X. It may also include alphabets 'A through F' or 'a through f'. The letters A through F represent the numbers 10 through 15. For example,

0X3 0x7F 0x

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. Larger integer constants can be stored by appending qualifiers such as U, L and UL to the constants. For example,

6587U or 6587u	(unsigned integer)
123456789UL or 123456789ul	(unsigned long integer)
8796531L or 87965311	(long integer)

(b) Real Constants: Sometimes integer numbers are inadequate to represent quantities that vary continuously such as price, temperature, heights, and so on. Fractional parts are used to represent these quantities. Such numbers are called real or floating point constants. For example,

127.90 89.78 –0.76 +457.0

These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part. Digits before the decimal point or after the decimal point can be omitted. For example,

.78 145. –.34 +.23

are all valid real numbers.

Real number may also be expressed in exponential (or scientific) notation. Example, the value 341.56 may be written as 3.4156e2 in exponential notation. e2 means multiply by 10^2. The general form is

mantissa e exponent

The mantissa part represents a real number expressed either in decimal notation or an integer notation. The exponent part represents an integer number with an optional plus or minus sign. The letter e can be written either in lowercase or uppercase. This notation represents a real number in floating point form. For example,

0.87e3 1.8e+3 4.5E-2

The numbers that are either very large or small in magnitude can be expressed in exponential notation. For example, 870000000 may be written as 8.7E8 or 87E7, -0.000000567 is equivalent to -5.67E-7.

7.6.2 Character Constants

(a) Single Character Constants: A single character enclosed within a pair of single quote marks is defined as single character constant. For example,

‘s’ ‘M’ ‘7’

In the above example, the character constant ‘7’ is not the same as the number 7. It takes integer values known as ASCII values.

For example, the statement

printf(“%d”, ‘b’);

would print the number 98, which is the ASCII value of the letter b. Similarly the statement

printf(“%c”, ‘98’);

would output the letter ‘b’.

(b) String Constants: A sequence of characters enclosed in double quotes is defined as string constants. The characters may be letters, numbers, special characters and blank space. For example,

“hi” “Hello” “M”

The character constant ‘M’ is not equivalent to the single character string constant “M”. Also the a character constant has an integer value whereas single character string constant does not have an integer value

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\b' stands for backspace character. These character constants are also known as escape sequences.

Table 7.5 *Backslash Character Constants*

Constant	Meaning
'\a'	Audible alert
'\b'	Back space
'\f'	Form feed
'\n'	New line
'\r'	Carriage return
'\t'	Horizontal tab
'\v'	Vertical tab
'\''	Single quote
'\"'	Double quote
'\?'	Question mark
'\\'	Backslash
'\0'	Null

7.7 VARIABLES

A data name that may be used to store a data value is defined as variable. A variable takes different values at different times during the execution.

Rules for naming the variables:

1. A variable name can be any combination of alphabets, digits or underscore.
2. They must begin with an alphabet. Some systems permit underscore as the first character.
3. The length of the variable should not exceed 8 characters.
4. No commas or blank spaces are allowed within a variable name.
5. Among the special symbols, an underscore can be used in a variable name.
6. Uppercase and lowercase letters are significant.
7. It should not be a keyword.

Examples:

```
Sum  total
```

7.8 DATA TYPES

C supports different data types and each data type may have predefined memory requirement and storage representation. 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.

ANSI C supports three classes of data types:

1. Primary or fundamental data type
2. Derived data type
3. User-defined data type

There are five fundamental data types namely integer (int), character (char), floating point (float), double precision floating point (double) and void.

Table 7.6 *Primary Data Types*

Data Types			
Primary	User Defined	Derived	Empty
Character char signed char unsigned char Integer **Signed** / **Unsigned** int / unsigned int short int / unsigned short int long int / unsigned long int Float float | double | long double	Typedef	arrays pointers structures union	Void

Table 7.7 *Size and range of basic data types on 16-bit machine*

Data Type	Description	Memory Bytes	Range of Values	Control String
int	Integer Quantity	2 Bytes	–32,768 to 32,767	%d or %i
char	Single Character	1 Byte	–128 to 127	%c
float	Floating point no.	4 Bytes	3.4 e-38 to 3.4 e+e38	%f
double	Double precision floating point nos.	8 Bytes	1.7e-308 to 1.7 e+308	%lf

Integer Types: Integer data types occupy one word of storage and the size of an integer that is to be stored depends on the computer. For a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767. A signed integer uses one bit for sign and 15 bits for the magnitude of the number. For a 32 bit word length, an integer can store values ranging from -2,147,483,648 to 2,147,483,647.

C has three classes of integer storage, namely, short int, int and long int in both signed and unsigned forms. **short int** represents small integer values and requires half the amount of storage as a regular **int** number. Unsigned integers use all the bits for the magnitude of the number and are always positive. The range of unsigned integer numbers will be from 0 to 65,535.

Floating point types: Floating point numbers (or real numbers) are stored in 32 bits with 6 digits of precision. Floating point numbers are defined by the keyword **float.** If the **float** type is not sufficient to store the number, the data type double can be used to define the number. A double data type uses 64 bits giving a precision of 14 digits. These numbers are known as double precision numbers and have greater precision than **float** type. **Long double** has extended precision up to 80 bits.

Void Types: This type has no values and is generally used to specify the type of function. The function is said to be void when it does not return any value to the calling function.

Character Types: Character data types are defined using the keyword **char.** Characters generally occupy 8 bits of storage. The qualifiers **signed** or **unsigned** are also applied for character data type. **signed chars** range between -128 and 127 and **unsigned chars** range between 0 and 255.

7.8.1 Variable Declaration

Any variable to be used in the program is to be declared before it is used in the program. Declaration of variables does two things:

1. It defines the variable name.
2. It specifies the data type of the variable.

Primary Type Declaration

<datatype> v1,v2,v3,……,vn;

where

datatype – type of data (e.g., int, char etc.)

v1,v2,v3,….vn- the list of variables

Variables are separated by commas and the declaration statement must end with a semicolon.

Examples:

```
int total;
double rate;
```

EXAMPLE PROGRAM

```
main( )
{
/* variable declaration */
   int m, n;
   int p;
   float f;
   /* actual initialization */
   m = 50;
   n = 40;
   p = m + n;
   printf("value of p : %d \n", p);
   f = 70.0/3.0;
   printf("value of f : %f \n", f);
}
The output is
value of p : 90
value of f : 23.333334
```

Initializing variables

It means providing an initial value to the variables. It is done using assignment operator.

Description

Datatype variable=constant;
Example int b=90
or
Variable=constant;
Example a=10, a=b=87

7.9 SCOPE OF THE VARIABLES

Scope of the variable is defined as the availability of variables within the program. There are two types of variables:

7.9.1 Local Variables

The variables which are defined within a function are called local variables and they can't be accessed outside the function.

EXAMPLE PROGRAM

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

```
int x = 45, y = 78;
/*x and y are local variables which are visible within this main function only.
and not visible to test funtion.*/
printf("\nvalues : x = %d and y = %d", x, y);
test();
}
void test()
{
int a = 100, b = 200;
/*a and b are local variables having scope within test function only.
they are not visible to main function.*/
printf("\nvalues : a = %d and b = %d", a, b);
}
```

Output

```
Values: x=45 and y= 78
Values: a=100 and b= 200
```

In the above example, x and y variables are visible within the main function only and are not visible to test function. Likewise, a and b variables are having scope within the test function only and are not visible to main function.

7.9.2 Global/External Variables

The variables that are available throughout the program are defined as global variables. Global variables are defined outside the main function, so that these variables can be accessed from anywhere in the program.

EXAMPLE PROGRAM

```
#include<stdio.h>
void test();
int m = 122, n = 144;
int a = 150, b = 180;
int main()
{
printf("All variables are accessed from main function");
printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b);
test();
}
void test()
{
printf("\n\nAll variables are accessed from test function");
printf("\nvalues: m=%d:n=%d:a=%d:b=%d",m,n,a,b);
}
```

```
Output
All variables are accessed from main function
Values: m = 122 : n = 144 : a = 150 : b = 180
All variables are accessed from test function
Values: m = 122 : n = 144 : a = 150 : b = 180
```

7.10 USER DEFINED TYPE DECLARATION

C supports user defined type declaration known as type definition that allows users to define an identifier that would represent an existing data type. The user defined data type identifier can then be used to declare variables. The general form is given by:

```
typedef type identifier;
```

type refers to the existing data type which may belong to any class of type. Identifier refers to the "new" name given to the data type but not the data type. **typedef** cannot create a new type. For example,

```
typedef int marks;
typedef float total;
```

where, **marks** symbolizes **int** and **total** symbolizes **float.** They can be used to declare the variables as follows:

```
marks m1,m2;
total name1[40],name2[40];
```

m1 and m2 are declared as **int** variables and name1[40], name2[40] are declared as 40 element floating point array variables.

The **typedef** creates meaningful data type names and increases the readability of the program.

Another user defined data type is enumerated data type. It is defined as follows:

```
enum identifier [value 1,value 2,…value n];
```

identifier is a user-defined enumerated data type that can have one of the values enclosed within the braces. These values are known as enumeration constants. After this definition, the variables of this 'new' type are declared as follows:

```
enum identifier v1,v2,…vn;
```

The enumerated variables v1, v2,…vn can have one of the values **value 1,value 2,…value n.**

Example:

```
enum day {Monday, Tuesday,…Sunday};
enum day week_st,week_end;
week_st=Monday;
week_end=Friday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value 1 is assigned 0, value 2 is assigned 1 and so on. This automatic assignment can be over-ridden by assigning values explicitly to the enumeration constants.

Example:

```
enum day{Monday=1,Tuesday,…Sunday};
```

In this example, Monday is assigned 1 and the remaining constants are assigned values that increase successively by 1.

The definition and declaration of the enumerated variables can be combined in one statement.

```
enum day {Monday, Tuesday,….Sunday} week_st,week_end;
```

EXAMPLE PROGRAM

```
#include<stdio.h>
enum new_enum{sunday,monday,tuesday,wednesday,thursday,friday,saturday};
const char *enum_names[]={"sunday","monday","tuesday","wednesday","thursday",
"friday","saturday"};
int main()
{
   enum new_enum e1,e2;
   e1=sunday;
   e2=monday;
   printf("day1=%s\n",enum_names[e1]);
   printf("day2=%s\n",enum_names[e2]);
   return 0;
}
```

Output

```
day1=sunday
day2=Monday
```

Declaration of Storage Class

Every variable in C programming has two properties: type and storage class. Type refers to the data type of a variable and, storage class determines the scope and lifetime of a variable.

There are 4 types of storage class:

1. Automatic
2. External
3. Static
4. Register

The storage class is another qualifier that can be added to a variable declaration as shown below:

```
auto int count;
register char ch;
static int x;
extern long total;
```

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as 'garbage') unless they are initialized explicitly.

Table 7.8 *Storage Classes and their Meanings*

Storage Class	Meaning
Auto	Local variable known only to the function in which it is declared. Default is auto.
Static	Local variable which exists and retains its value even after the control is transferred to the calling function.
Extern	Global variable known to all functions in the file.
Register	Local variable which is stored in the register.

7.11 ASSIGNING VALUES TO VARIABLES

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

variable_name = value;

Example:

Total = amount + inrate* amount;

In the above statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in the variable **amount.** The result is then added to the variable **amount** and the final value is stored in the variable **total.** The variable **value** is called the target variable. The variables that are used in the right side of the expression must be assigned values before they are encountered in the program.

7.11.1 Assignment Statement

The assignment statement has the following form:

variable = expression

In assignment statement, the expression on the right side is evaluated and then the value is assigned to the variable. The value assigned to the result must be of the same type as the result. The expression on the right side can be a single constant, a single variable or involve variables and constants combined by the arithmetic operators. The order of evaluation in matched pairs in expressions may be indicated using rounded brackets ().

+ addition
– subtraction
* multiplication
/ division
% remainder after division (modulus)

For example,

```
a = 10;
total = 1000.0;
x = 2.0 * (length + breadth);
ratio = (x + y)/(e + f);
```

Here are some rules:

- ❖ First the expression is evaluated.
- ❖ If the type of the expression and the variable are identical, then the result is saved in the variable.
- ❖ If not, the result is converted to the type of the variable and saved there.
  - If the data type of the variable is INTEGER while the data type of the result is REAL, to make it an integer result, the fractional part including the decimal point, is removed.
  - If the variable is REAL while the result is INTEGER, then a decimal point is appended to the integer making it a real number.
- ❖ The original one disappears once the variable receives a new value, and is no more available.

7.11.2 Compound Assignment

The compound assignment operators combine the simple assignment operator with another binary operator. It performs the operation specified by the additional operator and then the result is assigned to the left operand. For example, a compound-assignment expression such as

expression1 += expression2

can be understood as

expression1 = expression1 + expression2

Table 7.9 *Compound Assignment Operators*

Operator	Example	Meaning
+ =	c+=d	c=c+d
- =	e-=f	e=e-f
* =	m*=n	m=m*n
/ =	f/=g	f=f/g

Multiple assignments in one line are permitted in C. For example,

x=10;y=20;

are valid statements.

A value can also be assigned to a variable at the time the variable is declared. This takes the following form:

data-type variable_name=constant;

Example:

```
int x=100;
char y='x';
```

Initialization is the process of giving initial values to variables. More than one variable can be initialized in one statement using multiple assignment operators.

Example:

```
x=y=z=0;
p=q=s=MAX;
```

are valid. In the first statement, the variables x, y, and z are initialized to zero and in the second statement, the variables p, q and s are initialized with MAX which is a symbolic constant.

By default, external and static variables are initialized to zero. Automatic variables that are not initialized explicitly will contain garbage.

EXAMPLE PROGRAM

```
#include <stdio.h>
#include<conio.h>
void main()
{
int m,n;
clrscr();
m=10; //Assignment Statement
n=6;
m+=n; // Compound Statement
n-=m;
printf("\nThe value of m is %d",m);
printf("\nThe value of n is %d",n);
getch();
}
```

Output

```
The value of m is 16
The value of n is -10
```

7.11.3 Reading Data from Keyboard

Reading data from keyboard is another way of giving values to variables. This is achieved using **scanf** function which is a general input function in C. The general syntax of scanf is as follows:

scanf("control string", address of the variable)

The control string contains the format of the data being received. The ampersand symbol, '&', is an operator that specifies the variable name's address.

Example:

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

When this statement is encountered, the execution stops and waits for the value of the variable 'a' to be typed in. The control string '%d' specifies that an integer value is to be read from the terminal. Once the value is typed in and 'Return' key is pressed, the computer then proceeds to the next statement. The value is then assigned to the variable a.

Rules to be followed:

1. The control string must be given within double quotation.
2. The variables can be separated using comma.

3. It should be terminated by a semi colon.
4. No empty space should be given between two control strings.

Sample program using scanf() and printf() functions:

```
#include<stdio.h>
void main()
{
   int a;
   printf("\nEnter the value for a');
   scanf("%d",&a);
   printf("\n The value entered is %d",a);
}
```

Output

```
Enter the value for a 10
The value entered is 10
```

7.12 DEFINING SYMBOLIC CONSTANTS

A symbolic constant is a name that is substituted for a sequence of characters that cannot be changed and are defined in the beginning of the program. The character may represent a numeric constant, a character constant, or a string. Each occurrence of a symbolic constant is replaced by its corresponding character sequence when the program is compiled. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc.

For example, C program consists of the following symbolic constant definitions:

```
#define PI 3.141593
#define TRUE 1
#define FALSE 0
```

#define PI 3.141593 defines a symbolic constant PI whose value is 3.141593. When the program is preprocessed, all occurrences of the symbolic constant PI are replaced with the replacement text 3.141593.

The preprocessor statements begin with a #symbol and do not end with a semicolon. By convention, preprocessor constants are written in UPPERCASE.

There are two problems in using symbolic constants. They are

1. Problem in program modification.
2. Problem in program understanding.

Modifiability: If the value of the constant needs to be changed due to some reasons, then it has to be searched throughput the program and has to be changed explicitly wherever used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability: If a numeric value appears in a program, its use is not always clear, especially when the same values mean different things at different places. For example, the number 35 may represent

the number of students at one place, and the 'pass marks' at another place of the same program. The meaning of the variable may be confused when the program is executed some days later.

Assignment of such constants to a symbolic name avoids this problem. For example, the name **STRENGTH** may be used to define the number of students and **PASS_MARK** to define the pass marks. Constant values are assigned to these names at the beginning of the program. These names when used in the program are automatically substituted at the appropriate points. A constant is defined as follows:

#define symbolic-name value of constant

Example:

```
#define PI 3.141593
```

Symbolic names are sometimes called constant identifiers. Since the symbolic names are constants, they are not defined in declaration part. The following rules apply to a #define statement which defines a symbolic constant.

1. Symbolic names have the same form as variable names. They are written in capitals to distinguish them from the normal variable names which are written in lowercase letters.
2. No blank space between the pound sign **#** and the word **define** is permitted.
3. The first character in the line must be **'#'.**
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.
6. After definition, the symbolic name should not be assigned any other value within the program by using assignment statement.
7. Symbolic names are not declared for data types.
8. **#define** statements may appear anywhere in the program but before it is referred in the program.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
#define ON 1
#define PI 3.141593
void main()
{
float a;
float b;
float c;
float d=PI;
clrscr();
if(ON)
{
a=100;
b=a*10;
```

```
c=b-a;
}
printf("\na=%f\nb=%f\nc=%f\nPI=%f",a,b,c,d);
getch();
}
```

Output

```
a=100
b=1000
c=900
PI=3.141593
```

7.12.1 Declaring a Variable as a Constant

The variables needed to remain constant during the execution of a program are declared with the qualifier const at the time of initialization.

The syntax is as follows:

const type variable = value;

Example:

```
const int a=100;
```

where, **const** is a new data type qualifier. This statement says that the value of the **int** variable **a** must not be modified by the program. This variable can be used on the right side of an assignment statement like any other variable.

EXAMPLE PROGRAM

```
#include<stdio.h>
int main() {
const int LEN = 10;
const int WID = 5;
const char NEWLINE = '\n';
int area;
area = LEN * WID;
printf("value of area : %d", area);
printf("%c", NEWLINE);
return 0;
}
```

Output

```
Value of area: 5
```

7.12.2 Declaring a Variable as Volatile

The variables that may be changed at any time by some external sources like hardware, the kernel, another thread etc., are represented using **volatile** keyword.

Syntax

The keyword **volatile** may be added before or after the data type in the variable definition to declare a variable as **volatile**. For instance both of these declarations will declare **foo** to be a volatile integer:

volatile int foo;
int volatile foo;

When a variable is declared as **volatile,** the compiler will examine the value of the variable each time it is encountered to check whether the value has been altered or not.

On the other hand, if the value must not be modified by the program while it may be altered by some other process, then the variable may be declared as both **const** and **volatile.**

Example:

```
volatile const int a=100;
```

Overflow and Underflow of Data

The condition that occurs when a calculation produces a result that is greater in magnitude than that which a given register or storage location can store or represent is called as overflow. Underflow is the condition in a computer program that can occur when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value representable as a normal floating point number in the target data type.

Example:

If there are two unsigned integer types each with the value of 2174843648 (a & b), then a + b = 4349687296. This value is larger than the maximum value that can be represented in an unsigned integer type. This is called an integer overflow.

Another example, unsigned int a,b; a=0 b=a-1 The value of b is -1. This is called an integer underflow because the value is below than the minimum possible value that can be stored.