INTRODUCTION TO C – A PRE-REQUISITE

1.1 ALGORITHMS

Computer solves a problem based on a set of instructions provided to it. A problem should be broken into several smaller steps for it to be transformed into a set of instructions. Such a set of statements/instructions is called as an algorithm. In other words, ***Complete, unambiguous procedure (set of instructions) for solving a problem in a finite number of steps is called as algorithm.***

Every algorithm must be complete and yield at least one output. Every statement in the algorithm must be clear and should not contain any ambiguity. For example,

Add 3 or 5 to x

The above statement is ambiguous as it is not clear whether to add 3 to x or to add 5 to x. Such statements must be avoided in the algorithm. Moreover, every algorithm must give the result in finite number of steps.

Algorithm normally consists of English-like statements. We have a bit-more structured way of writing algorithm, called as ***pseudo code.***

Example 1. Write an algorithm for finding sum of two numbers.

Algorithm:

```
Step 1. Start
Step 2. Read two values, say a and b
Step 3. Add a and b,  and store the result in another variable, say c
Step 4. Display the value of c.
Step 5. Stop
```

Pseudo code:

```
Step 1. Input a, b
Step 2. c ⟵  a + b
Step 3.  Print c
```

Example 2. Write an algorithm for finding biggest of two numbers.

Algorithm:

```
Step 1. Start
Step 2. Read two values, say a and b
Step 3. Compare a and b. Store the larger number in another variable, say big.
Step 4. Display the value of big.
Step 5. Stop
```

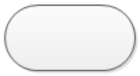
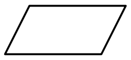
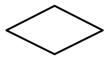
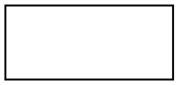
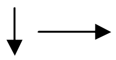
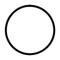
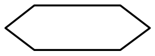
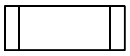
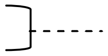
Pseudo code:

```
Step 1. Input a, b
Step 2.  if(a>b)
                big $\leftarrow$ a
          else
                big $\leftarrow$ b
Step 3.  Print big
```

NOTE: The student can choose either algorithm or pseudo code for solving a problem.

1.2 FLOW CHARTS

***Flow chart is a* pictorial representation of the algorithm.** Flow chart uses some geometrical shapes for representing an algorithm diagrammatically.

Symbol	Meaning	Symbol	Meaning	Symbol	Meaning
	Start/End		Input/Output		Decision
	Processing		Flow Lines		Connector
	Looping Statements		Pre-defined Process		Explanatory Notes

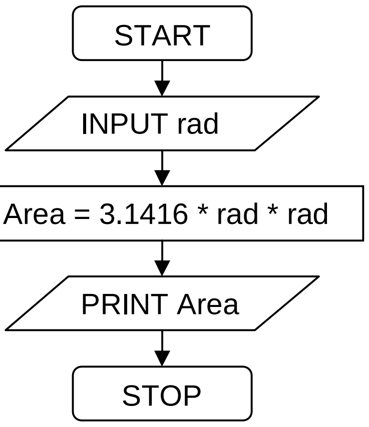
Advantages of using Flowchart:

- Flowchart acts as a blue print during program preparation phase. The program may be compared with flowchart to find if any statement is missing.
- Flowchart may be used to study different parts of a program to identify the problems and find an alternative approach.
- Flowchart helps in understanding the problem easily for any common man.
- In case a program has some logical errors, the flowchart helps in locating the error quickly and leads to easier debugging process.

Example: Write algorithm and flow chart for finding the area of a circle given the value of radius.

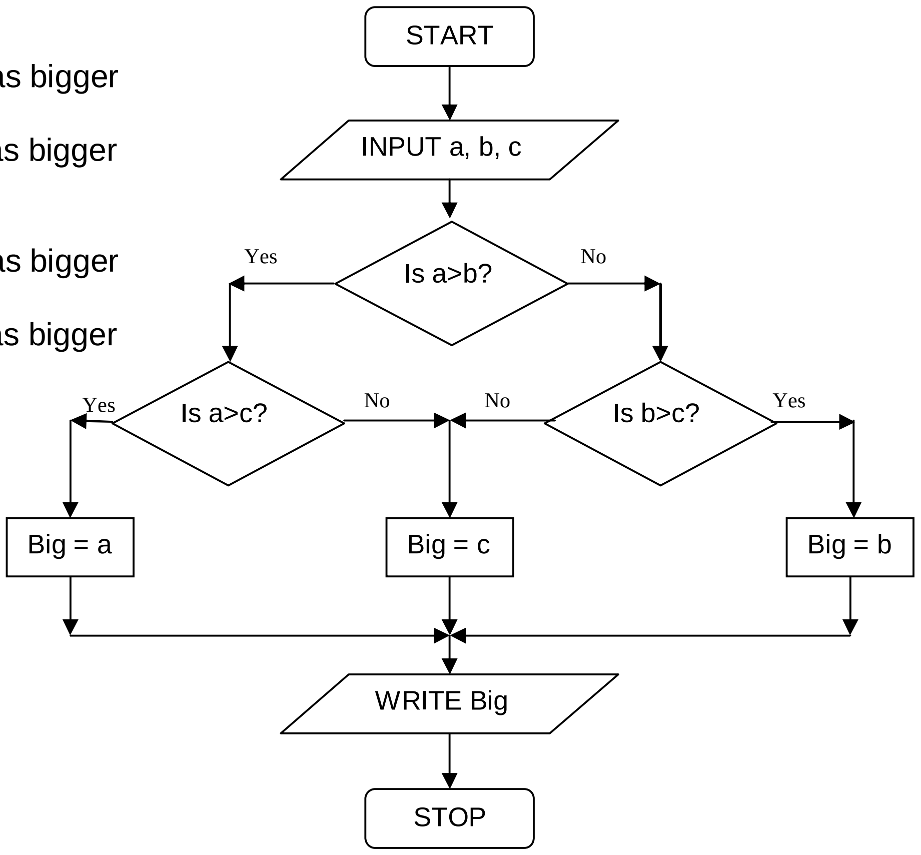
Algorithm/Pseudo Code

```
Step 1: Start
Step 2: Input the value of a radius say, rad.
Step 3: Calculate area = 3.1416 * rad * rad.
Step 4. Print the value of area.
Step 5: Stop.
```



Example: Write an algorithm and flow chart to find the biggest of three numbers.

```
Step 1: Read a, b, c
Step 2: if a>b then
                if a>c then
                        print a as bigger
                else
                        print c as bigger
        else
                if b>c then
                        print b as bigger
                else
                        print c as bigger
Step 3: Stop.
```



1.3 STRUCTURE OF C PROGRAM

The general structure of a C program can be given as -

```
#include<----->         }  Preprocessor Directives
#include<----->         }
#define -----------     }  Symbolic Constants/Macro Functions

Global Variable Declarations
User-defined Function Declarations/Prototypes

void main()
{
        Local Variable Declarations

        Program Statements
}

User-Defined function Definitions
```

Preprocessor is a software piece which scans the entire program before the source code is handed over to the compiler. Preprocessor directives can be used for including any header files like stdio.h, math.h etc. and also for declaring symbolic constants using #define statement. Preprocessor directives are discussed later in detail.

Global variables are the variables that can be used by all the function in the entire program. Such variable declarations can be provided after a list of preprocessor directives.

C is a modular programming language. That is, the given problem can be divided into several independent sub-problems. And each of these sub-problems can be solved independently. The solutions of all these can then be combined to get the solution for original problem. Such sub-problems can be solved with the help of **user-defined functions** (or modules). The declaration (return-type function-name (argument List)) can be done after/before global variable declarations. (Functions are discussed in later chapters in detail).

The C program execution always starts with a function main(). This function contains the declaration of local variables needed for the process inside the main() function. After local variable declaration, the main() function can contain program statements which may involve i/o operations, any process statements, function-calls etc.

User-defined functions declared earlier have to be defined with a specific set of operational statements later.

Note that, every statement in a C program is terminated by a semicolon and every block of code is enclosed within a pair of flower brackets.

1.4 CHARACTER SET

C programming language uses a finite set of symbols known as character set. A character may be printable or non-printable. The characters are useful in any language to construct identifiers and to form a valid statement. Every character is having an equivalent ASCII (American Standard Code for Information Interchange) code ranging from 0 to 255. Following is a list of printable characters in C.

Table 1.1 Character Set of C

Decimal Digits	**:** 0, 1, 2 … 9
Alphabets	**:** A to Z and a to z
Special Characters	**:** ! “ # $ % & ‘ (
	) * + - / . : ;
	< = > ? @ [ ] \
	^ _ , { } | ~

Note that ASCII codes for 0 to 9 is from 48 - 57,
for a to z is from 97 - 122
for A to Z is from 65 - 90

1.5 TOKENS

One or more characters grouped together to form a basic element in C is called as a token. A token can be a keyword, constant, variable, operator, string, special character etc.

Every word in C is classified as either identifier or keyword. Identifiers are used to name variables, functions, arrays, symbolic constants, macros etc.

1.6 KEYWORDS

Some words are used for particular purpose in C. They have a specific meaning and are reserved to do certain task. Such words are known as keywords or reserved words in C. There are 32 keywords currently defined in standard C as listed below.

Table 1.2 Keywords in C

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

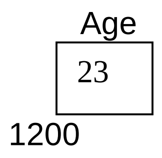
Some of these keywords are used for declaring type of variables, some for controlling the flow of statements etc. The usage of all these keywords will be understood in further discussion of the subject.

1.7 VARIABLES

A variable is a symbolic name to represent quantities in a program. In other words, variable is an identifier to name a specific memory location that can store the data. Computer identifies a memory location with an address assigned to it and each variable name in a program represents a memory location consisting of data. For example, the statement

int Age = 23;

implies that, Age is name of the variable having specific address and storing the value 23. It can represented as -



Here, Age is variable name, 1200 is the address of that variable and 23 is its value.

Note that, value of any variable is stored in binary format. That is, the value is converted into binary number and then it is stored. Also, the address will be actually in hexadecimal format and for the sake of simplicity, we will be considering in decimal format.

Rules for naming variables: To name a variable in C, certain rules have to be followed.

- Variable name should not be a C keyword.
- First character in a variable name must be any alphabet or an underscore (_).The remaining characters can be alphabets, digits or underscore.
- White (empty) space is not allowed.
- No special character other than underscore is allowed.
- The uppercase and lowercase alphabets are treated as different. For example, ***Age, age, aGe*** etc. are all treated as different variables. That is, *C is case-sensitive.*
- In most of the C compilers, the maximum number of characters allowed in a variable name is 8.

Following list gives a list of valid and invalid variable names –

Table 1.3 Example for valid and invalid variable names

Valid Names
Age
Test
Sum2
Stud_marks
_height
x2y2

Invalid Names	**Reason**
2sum	Should not start with digit
#age	Should not start with special character
height 3	Space is not allowed
Stud-marks	Hyphen (-) not allowed
$12Currency	Should not start with special character

NOTE:

- All the variables in a program must be declared with appropriate data type. The process of variable declaration assigns required number of memory locations (in bytes) for each variable based on data type.
- The data stored in a variable can be accessed just by specifying the name of that variable.
- The value of a variable can be changed by assigning new value to it or by reading from keyboard.
- The name of variable should reflect the meaning of value to be stored in it. For example, assume that age of a student is to be stored in a variable. Instead of naming it as *xyz*, it will be more meaningful if it is named as *stud_age* or simply *age*.

1.8 DATA TYPES

Basically, in C, the programmer can have four different types of data. That is, a value stored in a variable can be any one of four types viz. **int, float, char, double**. These are known as basic data types in C.

The integer (**int**) data type is used to store only integer values. The data type **float** is used store real numbers with fractional parts. Character (**char**) variables can store singe characters and **double** variables may store real numbers with higher precisions. These aspects are more elaborated in the following section.

1.9 CONSTANTS

A constant is a quantity that does not change during the program execution. Integer, floating point, character and the string are the four types of constants available in C. There are few modifiers like **short, long, signed, unsigned** etc. to provide various ranges in the values of constants. Each of the constant types are discussed here.

1.9.1 Integer Constants

These are the numeric values with no decimal point. An integer constant can be decimal, octal or hexadecimal. A decimal integer may consist of digits 0-9. An octal number may contain the digits 0-7 and is preceded by zero. Hexadecimal number consist of digits 0-9 and the alphabets *a* to *e* (or *A* to *E*). These are preceded by x or X. Following list gives some integer constants.

Table 1.4 Example for integers

Decimal	Octal	Hexadecimal
0	0	0
02	002	0x2
07	007	0x7
09	011	0x9
10	012	0xa
14	016	0xe

The range of values that can be stored in an integer variable depends on the word length of the computer. If *n* is word length of a computer, then the allowable range of numbers can be given by the formula

$$-2^{n-1} \text{ to } +2^{n-1}-1$$

For example, a 16-bit computer can store the integers in a range of -32768 to +32767. The highest bit (most significant bit) is always reserved for the sign of a number. If the most significant bit (MSB) is 1, then the number is negative, if MSB is 0, then the number is positive.

An integer data type can be either signed or unsigned.

- **Signed Integers:** The integers that may contain a sign-bit are signed-integers.
    - **int** It is a basic integer data type and requires minimum 16 bits (2 bytes).
    - **short int** It may be smaller than **int** or equal to **int** depending on machine. In some of the machines, **short int** will be half the size of **int.** If its size is 2 bytes, then the range of these numbers is -32768 to +32767.
    - **long int** It requires at least 32 bits. So, the allowed range is -2,14,74,83,648 to +2,14,74,83,647 (-2^{31} to $+2^{31}-1$).

- **Unsigned Integers:** In some of the programming situations, the programmer may need only positive constants. Then instead of wasting one bit for sign-bit, the programmer can go for unsigned numbers using the keyword **unsigned**. The unsigned numbers will make use of even the bit reserved for sign. Thus, the allowable range of unsigned integers will be 0 to 65635(0 to $2^{16}-1$). The programmer may use **unsigned short int** (2 bytes), **unsigned long int** (4 bytes) etc.

1.9.2 Floating Point Constants

The numeric values having decimal point and fractional part are known as floating point constants. For example - 8.75, 0.025, 123.89 etc. Since it is difficult to represent very large and very small floating point numbers in a standard decimal form, in C, the exponential form is used as –

(Mantissa) e (Exponent)

Here, Mantissa should have at least one digit along with decimal point and exponent may be either 2 or 3 digit integer. The following table gives some examples:

Table 1.5 Example of exponential notation

Decimal	**Exponential Form**	**C Exponential Form**
7653000	7.653×10^6	7.653e06
350000000	3.5×10^8	3.5e08
1000000000	1.0×10^9	1.0e9
0.000012	1.2×10^{-5}	1.2e-05
-0.0000034	-3.4×10^{-7}	-3.4e-07

The decimal part and fractional part of a floating point numbers are converted into binary format separately and then stored. Thus, the operations on floating point numbers are slower than that on integers.

To store floating point constants, three data types are available in C.

- **float** A float number has 6 digits after decimal point. It requires 4 bytes of memory and the allowed range of numbers is ***-3.4E48*** to ***+3.4E48***.
- **double** This types of constants are used to increase the precision. The number of digits in fractional part is 10 (It is Operating System dependent). It requires 8 bytes of memory and the range allowed is ***-1.7E308*** to ***+1.7E308.***
- **long double** This is used to store largest floating point number. The size is 10 bytes and the range of values is ***-1.7E4932*** to ***+1.7E4932.***

1.9.3 Character Constants

A single character enclosed within single-quotes is called as a character constant. Each character has an equivalent ASCII code. The binary equivalent of this ASCII code is stored with respect to a character constant. Thus, computer stores a character constant as an integer. Examples of character constants are -

‘M’, ‘#’, ‘9’, ‘r’, ‘1’ etc.

The size required for a character constant is 1 byte and the range of values is -128 to +127. Character can be **signed** or **unsigned.** A signed character is same as **char**, but unsigned character has the range from 0 to 255.

1.9.4 String Constants

A sequence of characters enclosed within double-quotes is known as string constant. For example,

“Hello, how are you?”
“Object Oriented Programming in C++”
“x”
“8” etc.

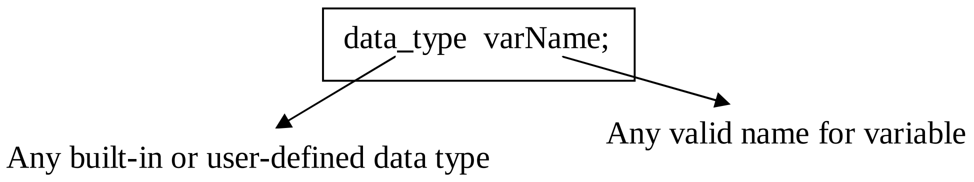
To indicate the end of a string, the compiler will append a null character (‘\0’) at the end of every string. Thus, the total size of any string constant will be total number of characters in a string plus one extra byte for null character.

Note that, “x” is different from ‘x’. The former is a string composed of x and null character requiring two bytes of memory. Whereas, latter is a single character having ASCII value 120 requiring single byte of memory.

String handling is discussed later in detail.

1.10 VARIABLE DECLARATION

In C, all the variables to be used in the program must be declared initially. The syntax of variable declaration is –


data_type varName;
Any built-in or user-defined data type
Any valid name for variable

For example,

```
int age;
float marks;
unsigned char gender;       etc.
```

Variable declaration informs the compiler about the type of the data (to allocate required number of memory in bytes) and about name of a variable.

The programmer can assign any value to a variable at the time of declaration itself. It is known as initialization of a variable. For example,

```
int age=26, salary=10000;
```

This kind of statements will serve both declaration and initialization of a variable.

1.11 SYMBOLIC CONSTANTS

The process of assigning a constant value to a variable may cause a problem during the program. Unknowingly the variable may get altered through assignment statement or any such other operation. To avoid this problem, C provides Symbolic Constants which are declared using preprocessor directive *#define.* The general form of declaring a symbolic constant is -

#define TAG   EXP

Here TAG is name of the symbolic constant and EXP is the value assigned to that constant.

For example -

```
#define PI 3.1416
#define MAX 5
```

Following are the rules to be followed while defining symbolic constants:

- There is no space between # and define, but there should be a space between ***define*** and *TAG* and *EXP.*
- The *#define* directive should not end with semicolon.
- Symbolic constant can not be changed later using any statement. That is,

```
#define MAX 5
……
……
MAX=3;       //error
```

- It is better to use capital letters for symbolic constants, to differentiate them from other variables of the program.
- Rules for naming symbolic constants are same as those of naming variables.

1.12 const QUALIFIER

In many programs, there are some situations where we need some value to be constant throughout the program. If we define such a value using any variable name as:

```
int size =10;
```

there is a chance that the value 10 of size may change. Because, after the initialization of size to 10, if the programmer himself unknowingly assigns some other value, say 15 as:

```
size =15;
```

size looses its original value. This may affect the logic of the program.

To prevent such changes in a constant value, we use a qualifier viz. *const*. For example:

```
const int size =10;
```

This keyword *const* assures that the value of the variable will not be changed throughout the program.

1.13 volatile QUALIFIER

This qualifier is used to tell the compiler that a variable's value may be changed, not explicitly specified by the program. For example, a global variable's address may be used to store the real time of the system. In this situation, the contents of the variable are altered without any explicit assignment statements in the program. Many of C/C++ compilers assume that a value of a variable is not changed if it does not appear at the left hand side of an assignment operator. Therefore, there is a chance that the program task is worked out without updating such global variables.

To prevent this from happening, the variable is declared as *volatile*, so that each time the external change occurs in the variable it will reflect in the program. Thus when a variable is preceded with the volatile qualifier, the compiler will not optimize the code using that variable.

It is always a good practice to declare a *volatile* variable with *const* qualifier. Because a volatile variable must not be changed by the program code.

For example:

```
volatile int disp_register;
volatile const int TIME;
```

1.14 COMMENT STATEMENT

Having comments in between a program code to indicate the purpose of a particular statement or function is a good programming style. Comment lines will not occupy space in the memory. The compiler ignores them before object code generation.

There are two types of comment delimiters in C. Whenever the comment text is more than one line, generally we use the pair /* and */. For example,

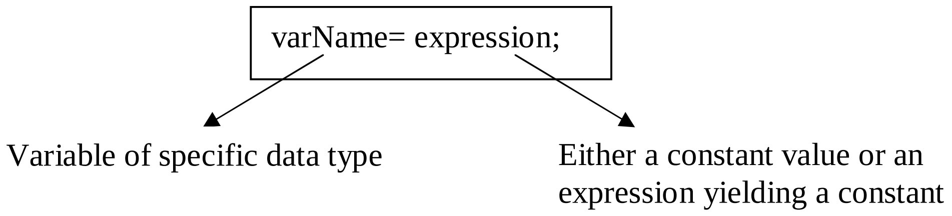
```
/* This is multi-line comment delimiter. A programmer can use
any of the comment delimiters as per programmer's
requirements. */
```

The above passage of text is ignored by the compiler when it is included in the C program, as it is written between /* and */. To comment a single line, one can use the second type of delimiter i.e. //. For example,

```
//This is single line comment.
```

1.15 ASSIGNMENT STATEMENT

It has been discussed that a variable name identifies a memory location and this memory location can store a constant value of a particular data type same as that of variable name. For any given moment of time, a variable can store a single constant value. To store a constant value into a variable, programmer has to use assignment statement. The syntax is –


varName= expression;
Variable of specific data type
Either a constant value or an expression yielding a constant

Here,

varName is known as **lvalue** and
expression is known as **rvalue**

Consider an example -

```
int a, b, c;
float x=2.5, y;
a=10;                    //constant value is assigned
b=20;
c=a+b;                   //evaluated expression is assigned
y=x;                     //value of one variable is copied to other
a=a+1;                   //value of a is increased by 1
```

1.16 INTRODUCTION TO OPERATORS

To solve any problem using computer, one may need to perform some calculations and various types of processes. C provides several types of operators to write a program involving calculations.

The symbols used to perform a specific type of operations are known as operators. The variables and/or constants upon which these operations are carried out are known as operands. An operator requiring two operands is known as **binary operator**, whereas, an operator requiring single operand is **unary operator.**

Different types of operators are discussed here under.

1.17 Arithmetic Operators

The symbols used to perform arithmetic operations are known as arithmetic operators. Following provides a list of arithmetic operators available in C.

Arithmetic operators

Type	Purpose	Operator	Syntax	Meaning
Binary Operators	Addition	+	a + b	Addition of a and b
	Subtraction	-	a – b	Subtract b from a
	Multiplication	*	a * b	Product of a and b
	Division	/	a / b	Divide a by b
	Modulus	%	a % b	Remainder after dividing a by b
Unary Operators	Positive	+	+a	Positive of a
	Negative	-	-a	Negative of a

A valid combination of variables and/or constants with arithmetic operators is known as ***arithmetic expression***. For example,

Assume, int a=10, b=4;

Then, a+b is 14
a-b is 6
a%b is 2
(a+b)/7 is 2 etc.

1.18 Increment and Decrement Operators

In many of the programming situations, the programmer has to increment and decrement value of a variable by 1. Programmer can follow traditional way like,

```
i=i+1; or
i=i-1;
```

But, C provides more simple way to achieve this task in the form of increment and decrement operators, written as ++ and --respectively. These operators can be used either as a prefix or as a postfix to the variable. The syntax is –

```
var++;
var--;
```
Post-increment/Post-decrement

```
++var;
--var;
```
Pre-increment/Pre-decrement

The usage of these operators is illustrated below.

Examples:

- int i=5;
  i++;   //value of i will be 6.

- int i=5;
  ++i;   //value of i will be 6.

- int i=10;
  i--;   //value of i will be 9

- int i=10;
  --i;   //value of i will be 9

- int i=5,j;
  j=i++;   //**i(=5) is assigned to j and then i becomes 6.**

- int i=5,j;
  j=++i;   //**i becomes 6 and then i(=6) is assigned to j.**

- int i=10,j;
  j=i--;   //**i(=10) is assigned to j and then i becomes 9.**

- int i=5,j;
  j=--i;   //**i becomes 9 and then i(=9) is assigned to j.**

Thus, it can be easily observed that, if post/pre increment/decrement operators are used with operand independently, it will not make any difference. But, if the incremented/decremented value is assigned to some other variable, then certainly, there will be change in the value of a variable at left-hand-side of assignment operator.

1.19 Compound Assignment Operators

Sometimes, the programmer has to perform some operation on a variable and the result must be stored in the same variable. For example,

x= x*5;

a= a+4;   etc.

In such situations, the assignment operator can be combined with arithmetic operators. Following is a list of compound assignment (short-hand) operators.

Compound Assignment Operators

Operator	Usage	Meaning
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Examples:

-
  ```
  int p=5;
  p+=3;          //(p = p+3) p is now 5+3 i.e. 8.
  ```

-
  ```
  int q=11;
  q%=2;          //(q = q%2) q is now 11%2 i.e. 1
  ```

1.20 Relational Operators

Operators used to identify the relationship between two operands are known as relational operators. The expressions involving these operators are relational expressions. Relational expressions always results in either true or false. Thus, they are also known as ***Boolean expressions.*** Following table gives the list of relational operators.

Relational Operators

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Examples:

-
  ```
  int a=5, b=10,c;
  c=(a>b);              //a>b is false. So, 0(false) is assigned to c.
  ```

-
  ```
  int x=10,y=10,z;
  z=(x==y);       //as x and y are equal, 1(true) is assigned to z.
  ```

1.21 Logical Operators

Some programming situations require that several relational expressions be evaluated and based on this result, the action should be taken. To combine relational expressions, C provides logical operators. Following is a list of logical operators.

Logical Operators

Type	Operator	Operation
Binary	&&	Logical AND
Binary	||	Logical OR
Unary	!	Logical NOT

The logical expressions results in either true (1) or false (0). Based on the truth-value of the operands, the expression yields the result. The truth-tables for these operators are given below:

Logical AND operation

Operand1	Operand2	Result
Zero	Zero	Zero
Zero	Non-Zero	Zero
Non-Zero	Zero	Zero
Non-Zero	Non-Zero	One

Logical OR operation

Operand1	Operand2	Result
Zero	Zero	Zero
Zero	Non-Zero	One
Non-Zero	Zero	One
Non-Zero	Non-Zero	One

Logical NOT operation

Operand	Result
Zero	One
Non-Zero	Zero

Examples:

-
  ```
  int i=10, j=20,k;
  k=i&&j;                //as i and j are non-zero values, k will be 1
  ```

-
  ```
  int i=10, j=0,k;
  k=i&&j;                //as one of the operands is zero, k will be 0
  ```

-
  ```
  int i=10, j=20, k=15, m;
  m= (i>j) && (j>k);

  /* The expression i>j is false. So, 0 is assigned to m */
  ```

-
  ```
  int i=10, j=20,k;
  k=i||j;          //as i and j are non-zero values, k will be 1
  ```

- int i=10, j=0,k;
  k=i||j; //as one of the operands is non-zero, k will be 1

- int i=10, j=20, k=15, m;
  m= (i>j) || (j>k);

  /* The expression i>j is false. But, j>k is true. As one of the operands for OR operator is true, 1 is assigned to m */

- int m=10,k;
  k=!m;

  /* 10 is true value. So, !m results to be false. So, 0 is assigned to k */

- int x=0,y;
  y=!x; //As x is false, y will be true (1).

1.22 Conditional (Ternary) Operator
When one among two situations must be opted based on some condition, the programmer can go for conditional operator. the syntax is –

```
var = (expr1)? (expr2): (expr3);
```

Here, expr1, expr2 and expr3 are any type of expressions and/or variables. expr1 is a condition resulting true or false. If expr1 results to be true, then expr2 will be evaluated and its value is assigned to var. Otherwise, evaluated result of expr3 is assigned to var. For example,

- int x=10,y=5,z;
  z=(x>y)?x:y;

  As the expression x>y is true here, value of x is assigned to z.

- float a=2.5,b=0.5,c;
  c=(a<b)?(a-b):(a+b);

  The expression a<b is false. So, (a+b) is evaluated and the result (2.5 +0.5 = 3.0) is assigned to c.

1.23 Comma Operator
To make the program compact, two or more distinct expressions can be combined to a single expression using comma operator. The syntax is –

```
(expr1, expr2, …, exprn);
```

Examples:

- int x=5,y=8, a=10,z;
  z=(++x, ++y, a);

  Here, both the expressions ++x and ++y are evaluated. So, values of x and y becomes 6 and 9 respectively. But the usage of comma operator within the pair of parentheses forces the last value to be assigned to a variable. thus, value of a (10) is assigned to z.

- int x=5,y=8, a=10,z;
  z=++x, ++y, a;

  Here also, x and y becomes 6 and 9 respectively. But, the expressions ++y and a are ignored for assignment and value of x (now, it is 6) is assigned to z.

1.24 Bitwise Operators

It has already been discussed that a computer can understand only zeros and ones (binary number format). Every program written by a programmer is translated into the form that a computer can understand and then only it gets executed. In C, programmer has a facility to operate on bits and thus having the features of low-level programming languages. The bit-wise operations provide an efficient way for interacting with the hardware and to perform some arithmetic operations in more elegant manner. Programmer can manipulate bits of variables through several bit-wise operators listed below –

Bit-wise Operators

Type	Operator	Meaning
Binary	&	Bit-wise AND
Binary	|	Bit-wise OR
Binary	^	Bit-wise XOR
Unary	~	One's complement
Binary	>>	Right-shift operator
Binary	<<	Left-shift operator

- **Bit-wise AND Operator:**
  As the name suggests, initially, operands are converted into binary-format. Then, the AND (&) operation is performed on the corresponding bits of operands. Consider an example –

  ```
  int x=5, y=6,z;
  z= x & y;
  ```

Now, this operation is carried out as –

```
x ⇨       0000 0000 0000 0101
y ⇨  &    0000 0000 0000 0110
          -------------------
z ⇨       0000 0000 0000 0100
```

Thus, z will be decimal equivalent of 0000 0000 0000 0100, which is 4.

NOTE:

- In the above example, it is assumed that an integer requires two bytes of memory. Hence, 16 bit values are considered.
- Whether a number is even or odd is decided by the value of a lower order bit in the corresponding bit pattern. If the last bit of a number is 1, then the number is odd. Otherwise, it is even. To check the value of lower order bit, one can use a specially called variable viz. **mask** having a value 1. Now, the number to be checked is ANDed (&) with **mask.** If the result is one, then number is odd. Otherwise, it is even. To illustrate this fact, consider an example:

```
int x=5, mask=1,y;
y= x & mask;
```

The operation is performed as –

```
      0000 0000 0000 0011
 &    0000 0000 0000 0001
      -------------------
      0000 0000 0000 0001,   which equivalent to decimal 1.
```

Thus, y will be 1 and so, x is odd.

Consider another example:

```
int x=10, mask=1,y;
y= x & mask;
```

The operation is performed as –

```
      0000 0000 0000 1010
 &    0000 0000 0000 0001
      -------------------
      0000 0000 0000 0000
```

Thus, y will be 0 and so, x is even.

- **Bit-wise OR Operator:**
  Here, the OR (|) operations is performed on individual bits of operands. For example –

```
int x=5, y=6,z;
z= x | y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0000 0000 0101
y ⇨   |   0000 0000 0000 0110
          -------------------
z ⇨       0000 0000 0000 0111
```

Thus, z will be decimal equivalent of 0000 0111, which is 7.

- **Bit-wise XOR Operator:**
  In XOR operation, if both bits are same (either both are 1 or both 0), then the resulting bit will be 0 (false). Otherwise, the resulting bit is 1 (true). For example –

```
int x=5, y=6,z;
z= x ^ y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0000 0000 0101
y ⇨   ^   0000 0000 0000 0110
          -------------------
z ⇨       0000 0000 0000 0011
```

Thus, z will be decimal equivalent of 0000 0011, which is 3.

- **One's Complement:**
  It is a unary operator. This operator changes all 1's of a binary number into 0's and vice-versa. For example –

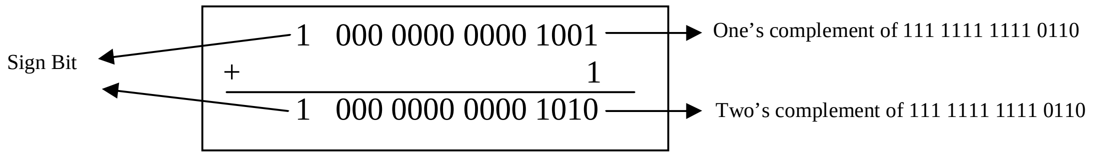
```
int x=9, y;
y= ~x;
```

Now, x = (0000 0000 0000 1001)
y = ~(0000 0000 0000 1001)
= 1111 1111 1111 0110

In binary form, we got the value of y as 1111 1111 1111 0110. It means, the sign bit (most significant bit) is 1 now. That, is the number is negative. Hence, while converting this number into decimal, the C compiler treats it as a negative number. And, negative numbers in C are represented as 2's complement. Note that,

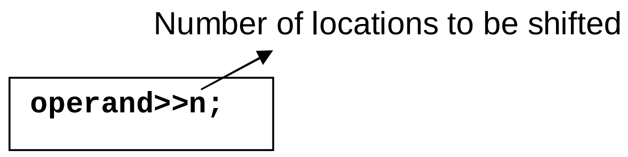
2's complement of a number = one's complement of that number + 1

Thus, in the above example, 2's complement of the number is taken by excluding sign bit. That is, 2's complement of (1111 1111 1111 0110) is calculated as –


Sign Bit
1 000 0000 0000 1001
+ 1
1 000 0000 0000 1010
One's complement of 111 1111 1111 0110
Two's complement of 111 1111 1111 0110

Now, this 2's complement is converted to decimal as -10. Hence, one's complement of +9 would be -10.

- **Right Shift Operator:**
This operator is denoted by >> (two greater-than symbols). It will shift each bit of the operand towards right through a number of positions as specified. And the empty bit-positions at left-side must be appended by zeroes. The syntax is:

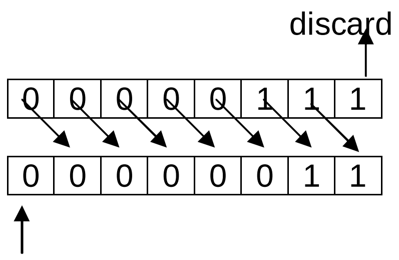


```
operand>>n;
```

Examples:

- int x=7,y;
y= x>>1;

  Here, the bits of x must be shifted one position towards right as shown –

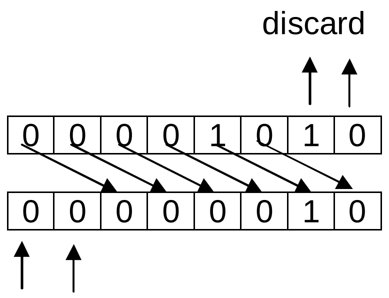


Append one zero at left.

Thus, y will be 3.

- int x=10,y;
y= x>>2;

  Here, the bits of x must be shifted two positions towards right as shown –



Append two zeroes at left.

Thus, y will be 2 now.

NOTE:

It can be easily observed that right-shift operation on an operand is equivalent to dividing the operand by 2^n. Thus,

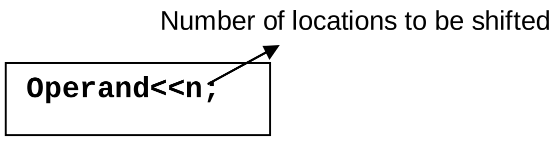
$$\boxed{\texttt{op>>n;} \quad \Longleftrightarrow \quad \texttt{op/}2^n}$$

Referring to above examples –

x>>1 is nothing but $x/2^1 = 7/2 = 3$ (integer division)

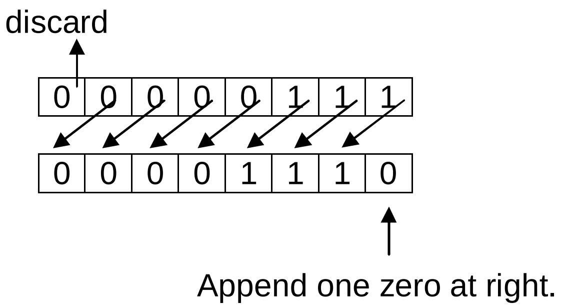
and x>>2 is(when x=10) $x/2^2 = 10/4 = 2$ (integer division)

- **Left Shift Operator:**
  This operator is denoted by << (two less-than symbols). It will shift each bit of the operand towards left through a number of positions as specified. And the empty bit-positions at right-side must be appended by zeroes. The syntax is:



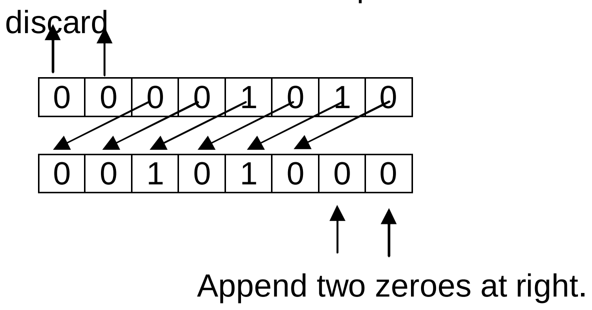
Examples:

- int x=7,y;
  y= x<<1;
  Here, the bits of x must be shifted one position towards left as shown –



  Thus, y will be 14.

- int x=10,y;
  y= x<<2;
  Here, the bits of x must be shifted two positions towards left as shown –



Thus, y will be 40 now.

NOTE:
Here also, one can observe that left-shift operation on an operand is equivalent to multiplying the operand by 2^n. Thus,

Op<<n; $\Longleftrightarrow$ **op*2^n**

Referring to above examples –
x>>1 is nothing but x*2^1 = 7*2 = 14 (integer division)
and x>>2 is(when x=10) x*2^2 = 10*4=40 (integer division)

1.25 Special Operators

There are some operators in C used for specific purpose as listed below:

- Address of Operator (&) : To extract the address of a particular variable.
- Value at the address (*) : To extract the value stored at a particular address (pointer)
- Dot operator (.) : To refer member variable of a structure/union/enumeration
- Indirectional Operaotr (->) : To refer member variable of a structure/union/enumeration
  through their pointers.

These operators will be discussed later in detail.

1.26 Precedence and Associativity of Operators

Computer evaluates several operations in an expression following a specific order known as precedence/hierarchy of operators. All the operators have two properties known as precedence and associativity. The associativity and precedence of various operators is listed in Table 1.12.

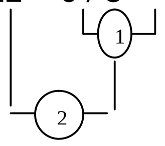
Operators with higher precedence have their operands bound or grouped to them before operators of lower precedence, regardless of the order in which they appear. For example, in the expression,

4+8*2

the multiplication (*) has higher precedence than addition (+). So, 8*2 is evaluated first and then 4 is added to 16.

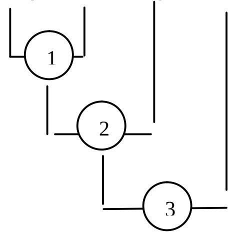
Associativity of an operator determines the direction in which the operands are associated with the operator. The association of operands with operator can be either from left to right or from right to left. The evaluation process of an expression is thus based on associativity of various operators involved in the expression. Consider the following examples:

- 12 + 6 / 3 = 14



- If two operators have equal precedence and occur one after the other, then they are evaluated in a sequence.

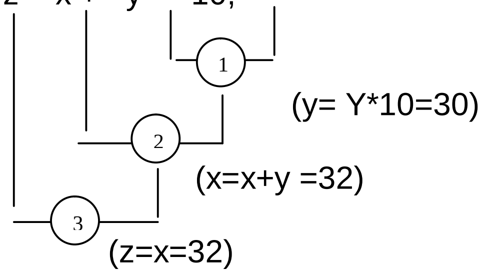
  12 / 4 * 2 + 5 = 11



  Here, / and * have equal precedence and have left to right associativity. Thus, they are evaluated in a sequence from left to right.

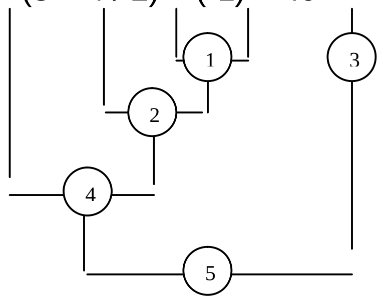
- int x=2,y=3,z;

  z = x += y *= 10;



  Here, the operators =, += and *= are all having same precedence. But, they are associated from right to left. Thus, the value of z will be 32.

- 10 * (3 + 4 / 2) + (-1) = 49



The parentheses have higher priority. So, all the operations within it are evaluated. Then the parenthesis containing unary operator – is evaluated. The next priority is the multiplication and finally addition. Note that all the operators involved here are associated from left to right.

Table 1.12 Precedence of Operators

Precedence	Operator	Operation	Associativity
1	()	Function Call/Parentheses	Left to Right
	[]	Array subscript	
2	!	Logical Negation(NOT)	Right to Left
	~	One's Complement	
	+	Unary Plus	
	-	Unary Minus	
	++	Pre/post increment	
	--	Pre/post decrement	
	&	Address of	
	*	Indirection	
	sizeof	Size of operand(in bytes)	
3	*	Dereference (pointer)	Left to Right
	->	Dereference (pointer to an object of a class or structure)	
4	*	Multiplication	Left to Right
	/	Division	
	%	Modulus (Remainder after division)	
5	+	Addition	Left to Right
	-	Subtraction	
6	<<	Left shift	Left to Right
	>>	Right shift	
7	<	Less than	Left to Right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
8	==	Equality	Left to Right
	!=	Not equal to	
9	&	Bit-wise AND	Left to Right
10	^	Bit-wise XOR	Left to Right
11	|	Bit-wise OR	Left to Right
12	&&	Logical AND	Left to Right
13	||	Logical OR	Left to Right
14	?:	Conditional (ternary) Operator	Left to Right
15	=	Assignment operator	Right to Left
	+=,*=, -=,/= etc.	Compound assignment operators	
16	,	Comma operator	Left to Right

1.27 Type Conversions

Some times it may happen that the type of the expression and the type of the variable on the left hand side of the assignment operator may not be same. In such cases, the value of the expression is promoted or demoted depending on the type of the variable on left hand side of =. Such kind of promotion/demotion of type is called as ***type conversion***. Since the compiler itself does this, it is also known as ***automatic/implicit type conversion.*** For example, consider a code segment:

```
int i;
float b;
i=3.5;
b=30;
```

Here, as *i* can not store fractional value, a floating number 3.5 is converted into integer 3. Similarly, 30 is promoted to 30.000000.

The same rule is applied for arithmetic expression also. For example,

```
float a=5.0, b=2.0;
int x;
x= a/b;
```

Here, x will get the value 2 instead of 2.500000.

Note that, always the promotion happens to the data type with more size.

Some times we need to force the compiler to explicitly convert the value of an expression to a particular data type. Consider a code segment,

```
float z;
int x=6, y=4;
z=x/y;
```

Here, as x and y are integers, the integer division is performed and x/y is evaluated as 1 instead of 1.500000. Now the integer 1 is promoted to 1.000000 and stored in the variable *z,* which is declared as float. Thus, we will not get the expected result. To avoid this problem, the programmer has to convert the type of data during evaluation of expression. This is known as ***explicit type conversion*** or ***type casting.*** The methodology is illustrated in the following code segment –

```
float z;
int x=6, y=4;
z=(float) x/y;
```

Here, the programmer is explicitly converting value of x from 6 to 6.000000. Then 4 is converted into 4.00000 implicitly by the complier. Thus, the value of expression is evaluated to be 1.500000.

1.28 Mathematical Functions

C provides some of the mathematical functions through the header file *math.h* for the programmer's usage. Following is a list showing few of the mathematical functions.

Function	**Meaning**
ceil(x)	x rounded ***up*** to the nearest integer
floor (x)	x rounded ***down*** to the nearest integer
exp(x)	e to the power of x
abs (x)	Absolute value of x
log(x)	Natural logarithm of x
log10(x)	Logarithm to base 10 of x
sqrt(x)	Square root of x
sin(x)	Sine of x
cos(x)	Cosine of x
tan (x)	Tangent of x
sinh (x)	sine hyperbolic of x
cosh (x)	Cosine hyperbolic of x
tanh (x)	Tangent hyperbolic of x

1.29 STORAGE CLASSES IN C

To understand the behavior of variables inside the function /block of code, we should understand the scope of variables. There is a concept called as *storage class* in C/C++, which will explain the storage area of the variable, default value of the variable, scope of the variable and life time of the variable. C programming language has four storage classes viz.

- Automatic storage Class
- Register storage class
- Static storage class
- External storage class

If user has not specified the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Usually, the default storage class will be *automatic storage class.* Following table briefly illustrates various storage classes.

Storage Class	**Storage Area**	**Default Value**	**Scope**	**Life-time**
Automatic (auto)	Memory	Garbage	Local to the block in which the variable is defined	Till the end of program
Register (register)	CPU Register	Garbage	- Do -	- Do -
Static (static)	Memory	Zero	- Do -	Value of the variable persists between function calls
External (extern)	Memory	Zero	Global	Till the end of program

Automatic storage class
The variables having this storage class is declared using the keyword 'auto'. Variables of this type are stored in the memory and their scope will be local to the block in which they are defined. The default initial value will be any garbage. These variables will alive till the end of the block in which they are defined.

Example:

```
int x =20;
void display()
{
        auto int a = 10;
        printf("%d",a);

        {
                auto int b =20;
                printf("%d",b);
        }
        printf("%d%d",a,x);
        printf("%d", b);      //this line generate error
}
```

Here, *a* is local to the function display(). That is, it can be accessed anywhere inside the function. But *b* is local to the block in which it is declared. So, it can not be accessed outside that block. The variable *x* is a global variable in this example, and it can be accessed anywhere in the program.

Note that, the keyword auto is optional. Even if we declare a variable inside the function as:

```
int a=10;
```

then, the compiler will treat that variable as auto only.

Register Storage Class
The declaration of register variables is done using the keyword register. Such variables are stored in CPU registers. They have the initial default value as any garbage value. The scope of these variables is being local to the block of their declaration. Life time is till the end of the block in which they are defined.

The number of CPU registers is limited for any system. Register variables may be treated as automatic variables when all the registers are busy doing some other task. Usually, the CPU registers in a microcomputer are of 16 bits, they can not hold float or double value. In this case too, the variables declared as register will be treated as auto. Thus, the declaration of variables as register is just a *request* to the compiler but not the *order.*

The variables declared as register will be accessed faster. It is customary to declare the most-frequently-used-variables as register. For example:

```
void disp()
{
        register int i;
        for(i=1; i<=10; i++)
                printf("%d",i);
}
```

The above function is used to display the numbers from 1 to 10. Here, the variable ‘i’ is declared as ‘register’. So, the variable ‘i’ is going to be stored in CPU register, if available.

Static Storage Class
The static variables are declared using the keyword static. They are stored in memory and initial default value is zero. The scope of the static variables is local to the block but the value of these variables persist between different function calls. For example:

```
void increment()
{
      /* here i is declared as static variable.  So, value of i is initialized when the
      function is called for the first time. Then for next calls, instead of re-declaration
      and re-initialization, the previous value is referred.*/

        static int i=0;
        int j =0;
        i++;
        printf("%d \t %d", i, j);
}
void main()
{
        increment();
        increment();
        increment();
    }
```

The output would be:

```
i=1     j=1
i=2     j=1
i=3     j=1
```

When we call the function increment() for the first time, obviously the value of i is printed as 1. However when we call the function for the second time, memory for the variable i is *not allocated* as it is declared as static. Thus, the previous value exists and is incremented. Hence we will get the output as 2. We get the output as 3 when the function gets called from the third time. But, the variable j is an auto variable. for each function call, memory for j will be re-allocated and initialized to 0. Thus, every time it will be zero only.

External Storage Class

Uses the keyword 'extern' for declaration. Default value being zero and stored in memory. Scope of these variables is global and so accessible for all the functions. The life of these variables exists till the end of the program. External variables can be used in two different contexts.

- In a single file containing C source code, an external variable can be treated as a normal global variable. for example:

```
#include<stdio.h>
int x=10;
void main()
{
        extern int x;
        printf("%d",x);
}
```

Here, the variable x is declared globally. So, when the declaration statement using *extern* keyword within function main() is encountered, the compiler will not allocate memory for x, but it uses the global variable x. Thus, external variable is just like a global variable in this situation.

- Consider one more situation now. Assume that there are two files of which one is containing just a variable declaration. And the other is containing source code and is required to use the variable declared in first file. This is shown below –

```
//file1.c
int x=10;
```

```
//file2.c
#include<stdio.h>
#include "file1.c"
void main()
{
     extern int x;
     printf("%d",x);
}
The output would be – 10
```

Here, the file **file1.c** must be compiled first. Then **file2.c** must be compiled and executed. Memory will be allocated for variable x only in the first file. When the compiler encounters the declaration statement with extern keyword in **file2.c**, then it searches for the variable x in all the included files. When it finds, the same will be used. If the programmer forgets to include the file **file1.c** in **file2.c**, then the compiler will generate the error.

NOTE:

- The usage of different storage classes should be made keeping the following issues in mind –
    - Economic usage of memory space
    - Improvement in speed of execution.

- Considering these two facts, user can use appropriate storage classes as under –
    - If there is a necessity of keeping values of variable between different function calls, use static.
    - If a particular variable, for ex., a counter variable in a loop, is used very often in the program, use register.
    - If a variable is used by many functions, use extern. Note that unnecessary usage of extern will lead huge wastage of memory.
    - If there is no need of above three situations, use auto.