DBIT16

ANNAMALAI UNIVERSITY

DIRECTORATE OF DISTANCE EDUCATION

BSC INFORMATION TECHNOLOGY

FIRST YEAR

PROGRAMMING IN C AND C++

Programming in C and C++

Table of contents

Unit IV

Unit V

Unit I

1.0 Objective

- Introduction to c
- Data Types
- Variables
- Constants
- Operators
- Control Structures
- Functions
- Arrays
- Structures
- Pointers

1.1 Introduction to C

C is a high-level programming language that forms the basis to other programming languages such as C++, PERL and Java. Dennis Ritchie developed it in 1972. He named it C because there was an existing programming language called B.

Programs are written in C. Computers don't understand C- the code needs to be compiled and turned into machine (binary) code before the program can be executed. Binary code is the lowest level language and can be thought of as consisting of 1's and 0's. One should also get a C Compiler to work with C. there are many free compilers available for C language.

C language opens the doors to other languages like C++, Java and even Java Script.

The C Character Set

The Character set consists of alphabet, digit or special symbol, that is a to z, A to Z, 0 to 9 and some special characters like underscore, hash, asterisk etc.

These basic characters are used to form constants, variables and keywords, equivalent to words in English. Keywords are words whose meaning has already been explained to the compiler or the computer. There are about 32 keywords in c and they are:

Auto	**double**	**if**	**static**
Break	**else**	**int**	**struct**
Case	**enum**	**long**	**switch**
Char	**extern**	**near**	**typedef**
Const	**float**	**register**	**union**
continue	**far**	**return**	**unsigned**
Default	**for**	**short**	**void**
Do	**goto**	**signed**	**while**
Sizeof	**volatile**		

Syntax

The C code written is called the SYNTAX.

Syntax is a mixture of:

- ✓ C Keywords like *int, for* and *return*
- ✓ Constants and Variables
- ✓ Operators like +(arithmetic "addition"), || (logical "or") and & (the "address of" operator).

Note that C is a case sensitive language, fro example, words like dog, Dog, dOg, doG are all considered different from one another.

Also the amount of "white space" used in a C program does not affect the way it's compiled. Use extra spaces can be used to make the programs more readable-indentation of code is very common. It is not allowed to put spaces or line breaks in the middle of keywords like this: *str uct*

Commenting Code

It is possible to add comments to the code by enclosing the remarks within /*and */. However, nested comments aren't allowed

A few properties of comments:

- ✓ They can be used to inform the person viewing the code what the code does. This is helpful when revisiting the code at a later stage.
- ✓ The compiler ignores all the comments. Hence, commenting does not affect the efficiency of the program.
- ✓ One can use /* and */ to comment out sections of code when it comes to finding errors, instead of deletion.

The *#include* Directive

If a line starts with a hash, denoted by #, it tells the compiler that a command should be sent to the ***C PREPROCESOR.*** The C Preprocessor is a program that is run where compiled. ***# include*** is one of the many C preprocessor commands.

Basically, when the preprocessor finds #include it looks for the file specified and replaces ***#include*** with the contents of that file. In away, this makes the code more readable and easier to, maintain if common library functions need to be used.

Header Files

Header Files have the extension .h and the full filename follows from the ***#include*** directive. They contain declarations to certain functions that may or may not have been used in a program.

For example, the stdio.h file is required if functions like ***printf*** and ***scanf*** are used in the program.

There are two ways to include a header file:

#include "stdio.h"
#include<stdio.h>

If the double quotes marks are used, it means that the directory currently in, will be searched for first for the header file, before any other directories are searched.

If the square brackets are used, directories other than the one currently in will be searched for the header file.

The *main* function

A *FUNCTION* can be thought of a set of instructions that's is carried out when it is *CALLED.* All C programs must have a main function. It is allowed to have only one, but can be placed anywhere within the code.

The program always starts with the main function and ends when the end of main is reached. Functions return a value too-this will be explained later. If a function returns nothing, its return type is of type *void*-that is nothing is returned.

The main function is special, as it returns an integer by default, which is why return 0; is written at the end of the program. Zero is usually returned to indicate error-free function termination. Another way to terminate a program is to use the *exit.*

1.2 Data Types

In C, there are four data types: char, int, float and double. Each one has its own properties. For instance, they all have different sizes. The size of a variable can be pictured as the number of containers / memory slots that are required to store it.

Declaring and Initializing Variables

To DECLARE a variable, means to reserve memory space for it. Declaring a variable involves inserting a variable name after a data type. It is also possible to declare many variables of the same data type al on one line by separating each one with a comma.

INITIALIZING a variable involves assigning (putting in) a value for the first time. This is done using the ASSIGNMENT OPERATOR, denoted by the equal sign, =

Declaration and initializing can be done on separate lines, or on one line.

The *Char* Data Type

Variables of the char data type can store a single character from a set of 256 characters.

All of the characters on a keyboard have a unique numerical code associated with it, so in reality, numerical codes are stored into char variables. The set of codes are known as ASCII codes, where ASCII stands for "American Standard Code for Information Interchange" and is usually pronounced "ask-ee".

Initializing Char Variables

To store a characteristic into a char variable, it should be enclosed with SINGLE quote marks. Double quote marks are reserved for STRINGS (an ARRAY of characters). It is possible to assign a char variable an integer, that is, the ASCII code. This example should help clarify the declaration and initialization of char variables. Ignore the printf function for now.

```
#include<stdio.h>
int main()
{/*this program prints Hello*/
char a,b,c,d;/* declare char variables*/
char e='o'; /* declaration and initialization*/
a='H'; /*initialize the rest…*/
b='e'; /* b=e is incorrect*/
c='I'; /* so is c='l' enclose the character with single quote marks*/
d=108; /*the ASCII code for l*/
```

```
printf("%c%c%c%c%c\n",a,b,c,d,e);
return 0;
}
```

The int Data Type

Variables of the int data type represent whole numbers. If a fraction is assigned to an int variable, the decimal part is ignored and the value is assigned is rounded down(or TRUNCATED) from the actual value. Also, assigning a character constant to an int variable assigns the ASCII value.

The float Data Type

To store variables correct to six decimal places, the float data type can be used. Floats are relatively easy to use but problems tend to occur when performing division. Generally
An *int* divided by an *int* returns an *int.*
An *int* divided by a float returns a *float.*
A *float* divided by an *int* returns a *float.*
A *float* divided by a *float* returns a *float.*

The *double* Data Type

It is possible to store decimals correct to ten decimal places using the double data type. However, doubles take up twice as much memory than floats, so double should be used when it's really necessary.

Type modifiers

The signed and unsigned keywords

When a variable of the type, int, by default is declared, its value is SIGNED. In other words, the variable could be positive or negative.

The minimum value is signed int is –32768 and the maximum value is 32767.
An unsigned int on the other hand, can only store positive values, and has the range from 0 to 65535.

The *short* and *long* keywords

The cases above apply when the integer is of the short type, which takes up less memory than the long type. The range of values that a short int could store is somewhat limited so if huge numbers are to be stored the long type should be used. Most of the time, an integer will be of the signed short type by default.

Here's a summary:

Type	Minimum Value	Maximum Value
Signed short int	-32768	32767
Unsigned short int	0	65535
Signed long int	-2147483648	2147483647
Unsigned long int	0	4294967295

The *sizeof* Operator
This function enbles to find out how many BYTES a variable occupies. A byte is defined as EIGHT BINARY DIGITS. The *sizeof* operator takes one *OPERAND.* An operand isa n expression that is required for an operator to work.

1.3 Variables

Variables are like containers in the computer's memory – one can store values in them and retrieve or modify them when necessary.

Constants are like variables, but once a valued is stored (i.e. the constant is *INITIALIZED*), its value cannot be changed.

Naming Variables

There are several rules that must be followed when naming variables:

Variable names	Example
CANNOT start with a number	2times
CAN contain a number elsewhere	times2
CANNOT contain any arithmetic operators...	a*b+c
... or any other punctuation marks...	#@%£!!
... but may contain or begin with an underscore	_height
CANNOT be a C keyword	while
CANNOT contain a space	stupid me
CAN be of mixed cases	HelloWorld

Local Variable

Local variable is the default storage for a variable in 'C' language. Its also called as automatic variables and represented by the keyword 'auto'. It should be declared at the start of the block.

When the program execution enters the block, memory is automatically allocated for this variable and released automatically upon exit from the block. The scope of automatic variable is local to the block in which they are declared. And they are so called *local variable*. We don't have direct access to local variables from outside the block but can access them indirectly using pointers.

External Variable

When a variable is declared outside the scope of any function that is outside the main function, then the variable is called as a global variable. Such variables are called *global variables*, and the C language provides storage classes which can meet these requirements; namely, the external and static classes.

Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name access the local variable cell.

Example:

```
int x,y;
float f;
char c;

main()
{

 }
```

We can also pre-initialize the variable as follows,

```
int x=4,y=3;
float f=7.23;
char c='Y';

main()
{

 }
```

Static Variable

static variable can't be reinitialized and it is the default storage class for global variables. The two variables below (x and y) both have a static storage class.

```
static int x;
int y;

{
   printf("%d\n", y);
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

'static' can also be defined within a function! If this is done the variable is initialized at run time but is not reinitalized when the function is called..

```
{
        static x=1;
}
```

Some Terminology

EXPRESSIONS consist of a mixture of constants, variables and operators. They return values.

Here are some examples of expressions:

```
17 /* a constant */
x  /* a variable */
x + 17/* a variable plus a constant */
```

STATEMENTS are instructions and are terminated with a semicolon';'. Statements consist of a mixture of expressions, operators, function calls and various keywords. Here are some examples of statements:

```
x = 1 + 8;
printf("We will learn printf soon!\n");
int x, y, z; /* more on "int" later */
```

STATEMENT BLOCKS, on the other hand, can contain a group of statements. The C compiler compiles the statement block as if it was just one statement. To declare a statement block statements should be enclosed between curly braces.

This example has a statement block in an if-else statement. Everything in each statement block is somewhat merged into a single statement. There are 2 statement blocks here:

Example:

```
        if(x==10)                              {/* block 1 */
          printf("x equals 10\n");

          x = 11;
          printf("Now x equals 11\n");

          x = x + 1;
          printf("Now x equals 12\n");
        }                                     /* end of block 1
*/

         else
        {                                      /* block 2 */
          printf("x not equal to 10\n");
          printf("Good bye!\n");
}                                    /* end of block 2 */
```

1.4 Constants

Types of Constants

Numbers are considered as *LITERAL* constants – It is not possible to change the number 20, nor assign something else into 20.

On the other hand, *SYMBOLIC* constants can be assigned a value during initialization and are represented by a word.

There are several ways to define constants in C. The `const` keyword is used to define a constant.

Suppose there is a circle with a fixed radius of 5 units. It is possible to create a symbolic constant like this:

const int radius = 5;

Since `radius` is declared using the `const` keyword, statements like: `radius = 12;` would be illegal.

Instructions

Instructions can be classified into four different types based on their purpose:

1. Type Declaration
2. Arithmetic instruction
3. Input / Output instruction
4. Control instruction

1.5 Operators

The *sizeof* Operator

This function enables to find out how many *BYTES* a variable occupies. A byte is defined as *EIGHT BINARY DIGITS (or "8-bits").* Binary numbers are covered later. For now, think of a byte as a container in the computer's memory.

The `sizeof` operator takes one *OPERAND.* An operand is an expression that is required for an operator to work.

A data type can be used as an operand, to find out how much memory is required to store variables of that data type, as demonstrated by this example:

Example:

```
#include <stdio.h>
int main()
 {
  printf("Size of int is %d bytes\n",sizeof(int));
  printf("Size of short int is %d bytes\n",sizeof(short int));
  printf("Size of long int is %d bytes\n\n", sizeof(long int));

  printf("Size of signed int is %d bytes\n",sizeof(signed int));
  printf("Size of signed short int is %d bytes\n",sizeof(signed short int));
  printf("Size of signed long int is %d bytes\n\n", sizeof(signed long int));

  printf("Size of unsigned int is %d bytes\n",sizeof(signed int));
  printf("Size of unsigned short int is %d bytes\n",sizeof(unsigned short int));
  printf("Size of unsigned long int is %d bytes\n\n", sizeof(unsigned long int));
    printf("Size of char is %d byte\n",sizeof(char));
  printf("Size of float is %d bytes\n",sizeof(float));
  printf("Size of double is %d bytes\n", sizeof(double));
 return 0;

  }
```

Notice the use of extra white space to add readability to the code.

The output shows that the signed or unsigned type modifiers do not affect the amount of memory required:

```
Size of int is 2 bytes

Size of short int is 2 bytes

Size of long int is 4 bytes
```

```
Size of signed int is 2 bytes

Size of signed short int is 2 bytes

Size of signed long int is 4 bytes

Size of unsigned int is 2 bytes

Size of unsigned short int is 2 bytes

Size of unsigned long int is 4 bytes

Size of char is 1 byte

Size of float is 4 bytes

Size of double is 8 bytes
```

It is also possible to pass the function a declared variable, to find out how much memory that variable occupies.

Arithmetic Operators

Arithmetic operators are commonly used in a variety of programming languages. In C, there are five of them, and they all take two *OPERANDS*. Recall that an operand is an expression that is required for an operator to work. For example, for `8 + 4`, `8` and `4` are considered as the operands.

Operator Name	Symbol
Multiplication	*
Division	/
Modulus	%
Addition	+
Subtraction	-

It is explained why the five operators are listed in this particular order...

The multiplication, division and modulus operators have higher *PRECEDENCE* over the addition and subtraction operators. This means that if an expression contains a mixture of arithmetic operators, multiplication, division and modulus will be carried out first in a *LEFT TO RIGHT* order, then any addition and subtraction.

Brackets (also known as *PARENTHESES*) can be used to change precedence, as everything enclosed within brackets is always evaluated first. For example, `2*4+3` returns `11` because `2*4` is `8`, and `8+3` is `11`. On the other hand, `2*(4+3)` returns `14` because `4+3` is `7`, and `2*7` is `14`.

What's With the % ?!

Multiplication, addition and subtraction are the simplest to use. Division is also easy, but care should be taken with the truncation of an *`int`* divided by an *`int`*. *casting* if necessary should be used. Now, the one that confuses novices is the modulus operator, sometimes known as the remainder operator.

To keep things simple, `a%b` returns the *REMAINDER* that occurs after performing `a/b`. For this operator, `a` and `b` MUST be integers!

For example, `6%3` returns `0` because 3 goes into 6 EXACTLY. Similarly, `4%4`, `8%2` and `16%8` all return `0`.

But `3%4` returns `3`. This is explained with an example: suppose if there are 3 holes to fill, but it is possible only to fill 4 holes at a time. It is not possible to fill a group of 4 holes; therefore the 3 holes are still empty. Similar idea applies for `7%4` because it is possible to fill in one group of 4 but still there are 3 holes remaining.

Note that anything modulus with zero returns infinity because anything divided by zero is infinite. In JavaScript, *`NaN`* means "Not a Number" - either it's infinity, or something daft is entered.

Anything modulus with itself is always zero because that number goes into itself exactly.

29%3 returns 2

27%24 returns 3

15%49 returns 15

4%26 returns 4

40%17 returns 6

Arithmetic Assignment Operators

Sometimes something like this is to be written: `x = x + 2;`

There is a better (and efficient) way of writing expressions like these by combining the operator with an equals sign, as shown in the table. Care should be taken with these though: `x *= y+z` is the same as `x = x*(y+z)` and NOT `x = (x*y) + z`.

Long Hand	Short Hand
`x = x * y;`	x *= y;
`x = x / y;`	x /= y;
`x = x % y;`	x %= y;
`x = x + y;`	x += y;
`x = x – y;`	x -= y;

Increment and Decrement Operators

Increasing an integer variable by 1 is a common process. There are people who write statements like `x+=1;` or even worse, `x=x+1;`

There is an easier way: `x++;` (POST-INCREMENT)

Alternatively, it is also allowed to use `++x;` (PRE-INCREMENT)

They both increase `x` by `1` (note that `x` MUST be an integer), They differ when it comes to statements like these:

```
y = x++;

y = ++x;
```

If `x` was `5`, `y = x++;` would assign `x` to `y`, THEN increase `x` by `1`. The end result is that `y` equals `5` and `x` equals `6`.

If `x` was `5`, `y = ++x;` would increase `x` by `1`, THEN assign `x` to `y`. The end result is that `y` equals `6` and `x` equals `6`.

Post-incrementing is done AFTER the assignment, pre-incrementing is done BEFORE the assignment.

Similar rules apply for decrementing. If `x is to be decreased` by `1` it is correct to write `x--;` (POST-DECREMENT) or `--x;` (PRE-DECREMENT), as opposed to: `x-=1;` or `x=x-1;`

Relational and Logical Operators

Relational Operators

Relation operators are used to compare numerical values. Here are the six relation operators:

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

The top four have higher precedence than the "equal to" and "not equal to" operators.

All arithmetic operators have higher precedence than the relation operators. The "equal to" operator should not be confused with the assignment operator.

Logical Operators

Conditions are expressions that can return one of two values: 1 (true) or 0 (false). Any non-zero value is considered true, where as zero is thought of as false.

Experiment with the relation operators:

The logical operators can be used to test for more conditions.

The first one is the NOT operator (some call it negation). It is denoted by the `!` symbol and takes one expression as its operand.

Then there is the AND operator, represented by `&&`, and the OR operator, denoted by `||` (double broken vertical bar symbols - shift backslash on the keyboard). Both of these operators require two expressions as their operands.

Now, their return values depend on the TRUTH VALUES of their operands - do they return 1 or 0? Logical operators are commonly summarized using TRUTH TABLES, which lists all the different truth combinations of the operands and the final logical operator truth-value. NOT reverses the truth value of its operand:

Expression	Return Value
!1	0
!0	1

AND returns 1 if both operands return non zero values (like 1):

Expression	Return Value
1 && 1	1
1 && 0	0
0 && 1	0
0 && 0	0

OR only returns 0 if both operands return zero:

Expression	Return Value
1 || 1	1
1 || 0	1
0 || 1	1
0 || 0	0

NOT has higher precedence than AND, which has higher precedence than OR.

Operator Precedence

A summary of the operators so far:
Highest Precedence
( ) !
* / %
+ -
< <= > >=
== !=
&&
||
Lowest Precedence

Here is an example to explain operator precedence. Take:

```
x+5 == 6
```

Since `+` has higher precedence than `==`, `x+5` is evaluated first, then the result is compared with `6`. But what about `x<3 && y<7 || y==5`?

Suppose `x` is `4` and `y` is `5`. Since the relation operators have higher precedence than the logical ones, the line can be rewritten as `0 && 1 || 1` because `4<3`, `5<7` and `5==5` return `0`, `1` and `1` respectively.

Now, `&&` gets evaluated first, then `||`.

`0 && 1` returns `0` and `0 || 1`, returns `1`.

This tends to get confusing, which is why brackets should be used when there are many operators on a single line. The following is easier to understand than the original:

```
(x<3 && y<7) || y==5
```

Expressions in brackets are evaluated first.

1.6 Control Structure

Control structure is used to control the sequence of the program flow. They are of two categories namely :

- Selection
- Loops

Selection

In this case a statement is executed based on whether a condition is true or false. Following statements are used in selective execution of statements.

The if Statement

The ***if*** statement is used for decision-making. It executes a statement based on the result of evaluation of an expression. The general form of the simple if statement is

```
if (expression)
    statement;
```

where ***statement*** may consist of a single statement or a set of statements. If the ***if*** expression evaluates to true, the ***statement*** or ***block*** following the ***if*** is executed. For example:

```
#include<stdio.h>
main()
{
 int a=10,b=10;
 if (b= =a)
        printf( " the 2 values are equal");
}
```

Output:
the 2 values are equal

In the above code since the value of a and b are the same, the expression in the if is true and hence the statement following the ***if*** expression is executed.

if …else

The ***if…else*** statement is used for decision-making. It executes a statement based on the result of evaluation of an expression. The general form of the if ..else statement is

```
if (expression)
    statement 1;
else
    statement 2;
```

where ***statement 1, statement 2*** may consist of a single statement or a block of statements. If the ***if*** expression evaluates to true, the ***statement 1*** or ***block*** following the ***if*** is executed; otherwise, the ***statement 2*** or ***block*** following the ***else*** is executed. The ***else*** statement is optional. It is used only if a statement or a sequence of statements is to be executed in case the ***if*** expression evaluates to false.

Consider the following Program:

```
#include <stdio.h>
main()
{
        int x, y;
        x = 3;
        y = 4;
        if (x > y)
          printf(" x is greater");
        else
          printf(" y is greater");
}
```

Output:
y is greater

In the above program, variable x has been assigned the value 3 and variable y has been assigned the value 4. In this case, since the value of x is not greater than y, the ***if*** expression is false and hence the statement following ***if*** is not executed while the statement following ***else*** is executed.

The if-else-if Statement

This is another common programming construct adapted from the initial ***if*** statement. The general form of this is:

```
if (expression) statement;
else
if (expression) statement;
else
if (expression) statement;
  …..
  …..
else statement;
```

This construct is also known as the ***if-else-if ladder*** or ***if-else-if staircase.***

Though the above indentation is easily understandable with one or two ***if****s*, it confuses as the number of ***if*** increases. This is because it gets deeply indented and so, the ***if-else-if*** is generally indented as:

```
if (expression)
        statement;
else if (expression)
        statement;
            .
else
            statement;
```

The conditions are evaluated from top to bottom. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true the final ***else*** is executed. If the final ***else*** is not present, no action takes place if all other conditions are false.

The following example takes a choice from the user. If the choice ranges from 1 to 3 it prints the choice else it prints ***Invalid choice***.

```
#include <stdio.h>
#include <conio.h>
main()
{
        int x;
        x = 0;
```

```
        clrscr();
        printf("Enter Choice (1 - 3) : ");
        scanf("%d", &x);
        if (x == 1)
                printf("\nChoice is 1");
        else if (x == 2)
                printf("\nChoice is 2");
        else if (x == 3)
                printf("\nChoice is 3");
        else
               printf("\nInvalid Choice ");
}
```

Output:

```
Enter Choice (1 - 3) : 1
Choice is 1
```

If the first condition ***(x ==1)*** is true then, the following ***printf()*** will be executed and the control will come out of the block. If the first condition is not true, the ***if*** corresponding to the first ***else*** will be checked. In case this ***if*** condition is satisfied, the ***printf()*** corresponding to this ***if*** will be executed and the block will be exited; otherwise the ***else*** following this ***if*** will be executed.

The switch Statement

The ***switch*** statement is a multi-way decision maker that tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of the ***switch*** statement is:

```
switch (expression)
{
        case constant1:
              statement sequence
              break;
        case constant 2:
              statement sequence
              break;
        case constant 3:
              statement sequence
              break;
        .
        .
        .
        default:
              statement sequence }
```

where ***switch, case, break*** and ***default*** are the keywords and statement sequence can be simple statement or a compound statement which need not be enclosed in parentheses. The expression following **switch** must be enclosed in parentheses and the body of the **switch** must be enclosed within the curly braces. The datatype of the expression given and the datatype of the case constants given should be compatible. As suggested by the name, ***case labels*** can be only integer or character constant or constant expressions, which do not contain any variable names***. 'Case'*** labels must all be different.

In the ***switch*** statement, the expression is evaluated and the value is compared with the ***case labels*** in the given order. If a ***label*** matches with the value of the expression, the statement mentioned will be executed. The ***break*** statement ensures immediate exit from the switch statement. If a ***break*** is not used in a certain ***case***, the statements in the following ***case*** are also executed irrespective of whether that ***case*** value is satisfied or not. This execution will continue till a ***break*** is encountered. Therefore, ***break*** is said to be one of the most important statement while using a ***switch***.

The statements against ***default*** will be executed, if none of the other cases are satisfied. The ***default*** statement is optional. If it is not present and the value of the expression does not match with any of the cases, then no action will be taken. The order of the ***case labels*** and default is immaterial.

```
/* switch Statement */
#include<stdio.h>
#include<conio.h>
main()
{
        char ch;
        clrscr ();
        printf("\n Enter a lower case alphabet (a-z): ");
        scanf("%c", &ch);
        if (ch < 'a' || ch >'z')
        printf("\nCharacter not a lower case alphabet");
        else
        switch(ch)
                {
                case 'a':
                case 'e':
                case 'i' :
                case 'o':
                case 'u':
                printf("\nCharacter is a vowel");
                break;
                case 'z' :
                printf("\nLast Alphabet (z) was input");
                break;
```

```
                default:
                printf("\nCharacter is a consonant");
                break;
                }
}
```

Output:

Enter a lower case alphabet (a – z): g
Character is a consonant

The program will take a lower case alphabet as an input and display whether it is a vowel, the last alphabet or a consonant.

Loops

Loops follow the iteration concept. Loops allow a set of instructions to be performed until a certain condition becomes false. This condition may be predefined or open ended. The loop structures available in 'C' are:

- The **for** loop
- The **while** loop
- The **do...while** loop

for loop

It is an entry-controlled loop that provides a more concise loop control structure. The general form is:

```
for (initialization; test-condition; increment/decrement)
        {
                statement
        }
```

The execution of ***for*** statement is as follows:

- ***Initialization*** of the control variables is done by using the assignment statement such as ***i***= 1 and ***count*** =0. The variable ***i*** and ***count*** are known as loop-control variables.
- The value of the control variable is tested using the ***test-condition***. The test condition is a relational expression, such as ***i*** < 10 that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise, the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the ***for*** statement after evaluating the last statement in the loop. Now, the control variable is ***incremented*** using an assignment statement such as ***i = i + 1*** and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Consider the following segment of a program:

```
for (x = 0 ; x<=9; x= x+1)
{
   printf("%d",x);
 }
printf("\n");
```

This ***for*** loop is executed 10 times and prints the digits 0 to 9 in one line. Semicolons must separate the three sections enclosed within parentheses. Note that there is no semicolon at the end of the increment section, ***x = x +1***.

```
/* Program to display 1 to 10 numbers on the screen */
#include <stdio.h>
main()
{
        int num;
        for(num =1 ; num <=10; num ++)
                printf("%d", num);
}
```

Output:

12345678910

The while Statement

The ***while statement*** is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.
The general form of while statement is

```
while (expression )
{
        statement
}
```

The ***statement*** will be executed repeatedly, as long as the ***expression*** is true. This statement can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition for the loop.

```
#include <stdio.h>
main()  /* display the integers 0 through 9 */
{
   int digit = 0;
```

```
    while (digit <= 9)
          {
            printf("%d\n",digit);
            ++digit;
           }
}
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Initially, digit is assigned a value 0. The ***while…loop*** then displays the current value of digit, increase its value by 1 and then repeats the cycle, until the value of digit exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning with 0 and ending with 9.

The do…while Loop

The ***do…while loop*** is sometimes referred to as ***do…loop*** in 'C'. Unlike ***for*** and ***while loops***, this loop checks its condition at the end of the loop, that is, after the loop has been executed. This means that ***do…while*** loop will execute at least once, even if the condition is false initially. The general form of ***do…while*** is:

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

The curly brackets are not necessary when only one statement is present within the loop, but it is a good habit to use them. ***do..while*** loop iterates until the condition becomes false. In the ***do…while*** the statement (block of statements) is executed first, then the condition is checked. If it is true, control is transferred to the ***do*** statement. When the condition becomes false, control is transferred to the statement after the loop. Consider the following:

```
/* accept only int values */
#include <stdio.h>
 main()
{
```

```
        int I,j;
        I = j = 0;
        do {
             printf("\nEnter : ");
             scanf("%d",&I);
             printf("No. is %d", I);
             j++;
        } while (I !=0);
        printf("\nThe total numbers entered were %d", --j);
        /* j is decremented before printing because count for last integer (0) is not to be
         considered */
}
```

Output:

```
Enter : 5
No. is 5
Enter : 3
No. is 3
Enter : 0
No. is 0
The total numbers entered were 2
```

The above program accept integers and display them until zero(0) is entered. It will then, exit the ***do…while*** and print the number of integers entered.

1.7 Functions

Introduction

A function can be thought of as a mini-program, where a group of statements are executed when the function is called.

A function is CALLED (or INVOKED) when it is needed to branch off from the program flow and execute a group of statements within that function. Once the statements in the function are executed, program flow resumes from the place where the function is called. A few functions: `main`, `printf` and `scanf were already used`.

The `main` function is special, in the way that it's called automatically when the program starts. In C, and other programming languages, it is possible to create customized functions.

A Typical Function

Functions have 5 main features:

1.	The RETURN TYPE is the data type of the RETURN VALUE of the function.
2.	The NAME is required as an identifier to the function, so that the computer knows which

	function is called. Naming of functions follows the same set of rules as the naming of variables.
3.	Functions can take ARGUMENTS - a function might need extra information for it to work. Arguments are optional.
4.	The function BODY is surrounded by curly brackets and contains the statements of the function.
5.	The RETURN VALUE is the value that is passed back to the main program. Functions exit whenever a value is returned.

Function Declaration

This is what a function definition might look like:

```
int squareNumber(int a)
{
int b = a*a;
return b;
}
```

`squareNumber` is the name of this function. Because an integer is returned, the `int` keyword must be placed before the function name. If the function does not return a value, the `void` keyword should be put before the function name.

This function has one argument, which is of the type `int`. If there are arguments, variable declarations should be in the round brackets.

The function body consists of 2 statements. The first, sees an `int` variable `b` declared and assigned `a*a`, i.e. `a` squared. The second statement uses the return keyword to pass the value of b back into the main program, hence exiting the function.

Within the program, it is possible to write:

```
x = squareNumber(5);
```

This would assign 25 to `x`. It is said that `5` is passed as an argument to the function `squareNumber`.

The variables within the `squareNumber` function are LOCAL VARIABLES - when the function exits, variables `a` and `b` are deleted from memory.

`int squareNumber(int a)` is also known as the FUNCTION HEADER.

Example :

```
#include <stdio.h>

void printAverage(int x, int y, int z);  /* the function declaration */

int main()
{
int a, b, c;

printf("Enter 3 integers separated by spaces: ");
scanf("%d %d %d", &a, &b, &c);

printAverage(a, b, c);   /* the function call */

return 0;              /* exit main function */
}
```

```
void printAverage(int x, int y, int z)
{                                   /* the function definition */
float average = (float) (x + y + z) / 3;   /* coercion in practice! */
printf("The average value of %d, %d and %d is %f\n", x, y, z, average);
}
```

It's common practice to place the function definition underneath `main` - if `main` is edited most of the time, as it is not needed to scroll too far down the page to get to it.

The function definition can be put above it if wanted. But if it is placed underneath `main`, make sure to put the function declaration above `main` - see the example. This is because the computer won't know if the function exists if called without it being declared in the first place. It's the same with variables: It is possible to assign anything to `x` unless `x` is declared beforehand. The function declaration is a single statement consisting of the function header - don't forget the semi colon at the end.

Notice that in the function call three arguments had to be passed to match the three arguments in the function definition. And the variable names in the function's argument section didn't have to match the variable names in the function call. The most important thing is that the data types had to match.

And notice the use of the `void` keyword for the `printAverage` function, since no value is returned.

Scope of Function variables

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

Local variables are declared within a function. They are created a new each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

Static variables are slightly different; they don't die on return from the function. Instead their last value is retained, and it becomes available when the function is called again. Global variables don't die on return from a function. Their value is retained, and is available to any other function, which accesses them.

Modifying function argument

Some functions work by modifying the values of their arguments. This may be done to pass more than one value back to the calling routine, or because the return value is already being used in some way. C requires special arrangements for arguments whose values will be changed.

You can treat the arguments of a function as variables, however direct manipulation of these arguments won't change the values of the arguments in the calling function. The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.

There is a way to change the values of variables declared outside the function. Passing the addresses of variables to the function does it. These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer. It is what lies at the address which the pointer references which interests us.

To get back to our original function, we pass it the address of a variable whose value we wish to change. The function must now be written to use the value at that address (or at the end of the pointer). On return from the function, the desired value will have changed. We manipulate the actual value using a copy of the pointer.

Functions That Call Other Functions

```
#include <stdio.h>

int triangular(int x);

int main()
{
int x;
```

```
printf("Enter an integer: ");
scanf("%d", &x);

if(x%10>3 || x==11 || x==12 || x==13)
{
printf("\n%d is the %dth triangular number\n", triangular(x), x);
}

else

{
switch(x%10)
 {
case 1:
printf("\n%d is the %dst triangular number\n", triangular(x), x);
break;

case 2:
printf("\n%d is the %dnd triangular number\n", triangular(x), x);
break;

case 3:
printf("\n%d is the %drd triangular number\n", triangular(x), x);
break;
 }
}
printf("You entered: %d\n", x);
return 0;
}


int triangular(int a)
{                              /* the nth triangular number is 1+2+3+ ... +n */
int x = (a * (a + 1)) / 2;
return x;
}
```

Above `main` is the function declaration. The `triangular` function takes an integer argument, and returns an integer.

Below `main` is the function definition - it works out and returns the triangular number of the number passed to the function.

Inside `main`, observe that inside `printf`, the `triangular` function is called. The `if`, `else` and `switch` blocks determine how to display the result.

One important thing to remember is that the `x` in `main` is totally different to the `x` in `triangular`.

The value of x in main remains unchanged after calling `triangular`.

Recursive Functions

Functions That Call Themselves

```
#include <stdio.h>
int factorial(int x);
int main()
{
  int a, b=1, temp;

  printf("Enter an integer: ");
  scanf("%d", &a);

  printf("\n%d factorial is %d\n\n", a, factorial(a));

  printf("Enter another integer: ");
  scanf("%d", &a);

 temp = a;

 for( ; a>0 ; a--)
 {
  b*=a;
 }
 printf("\n%d factorial is %d\n", temp, b);

 return 0;
}
int factorial(int x)
{       /* n factorial, (or n!) is 1*2*3* ... *n */
 if(x!=0)
 {
  return (x*factorial(x-1));
 }

 else
 {
```

```
  return 1;
  }
}
```

A function that calls itself is said to be a RECURSIVE function. Recursive function calls can be less efficient than a loop.

1.8 Array

An Array is a collection of variables of the same data type that are referenced by a common name. Each element in an array is associated with an index and occupies contiguous memory location. An array declared should indicate three things:

- Type of value to be stored in each element.
- The name of the array.
- The number of elements in an array.

Syntax

type_name identifier [Number of elements];

where ***type_name*** is the data type.

Example

char name[20];

The above example allocates memory for 20 character elements contiguously and is referenced by the identifier *name*

int numbers[50];

The above example allocates memory for 50 integer elements contiguously and is referenced by the identifier *numbers*.

Arrays can be basically classified into single dimension array and multi dimension array. Single dimension array has a single index while multi-dimension arrays have more than one index.

Initialization of Arrays

Array, a continuous stream of memory locations can be assigned value during declaration as shown in the following example.

Example

int numarray[5]={0,1,2,3,4}; /* This initialization is allowed */
int array[4]; /* This declaration is allowed */

array[4]={1,2,3,4};/* This assignment is not allowed */

However, use subscripts and assign values to the element of an array individually as follows, array[0]=1, array[1]=2, etc. Provide fewer values than the array size. In that case, the first part of the array is initialized. The following program illustrates how values are assigned to array elements and how they are retrieved.

```
#include<stdio.h>
main()
{
        int i;
        float x[5], value, total;


        printf("ENTER 5 NUMBERS \n");
        for (i=0; i<5; i++)
        {
                scanf("%f", &value);
                x[i]=value;
        }
         printf("\n");
                printf("The following are the numbers stored in the
array\n");
               for (i=0; i<5; i++)
               {
                  printf("x[%d]=%5.2f\n", i, x[i]);
               }
        }
```

Output:

```
ENTER 5 NUMBERS
1  2  3  4  5
The following are the numbers stored in the array
x[ 0]= 1.00
x[ 1]= 2.00
x[ 2]= 3.00
x[ 3]= 4.00
x[ 4]= 5.00
```

Character Arrays

So far arrays of integers are used. Arrays for `float`s and `double`s as well as `char`s can also be used.

Character arrays have a special property... Each element of the array can hold one character. But if the array is ended with the NULL CHARACTER, denoted by `\0` (that is, backslash and zero), it is called a STRING CONSTANT. The null character marks the end of a string - useful for functions like `printf`. Here is an example:

```
#include <stdio.h>

int main()
{
char charArray[8] = {'F','r','i','e','n','d','s','\0'};
int i;

for(i=0 ; i<8 ; i++)
{

printf("charArray[%d] has a value of %c\n", i, charArray[i]);
}
printf("My favourite comedy is %s\n", charArray); /* Alternative way */

return 0;
}
```

Output:

```
charArray[0] has a value of F

charArray[1] has a value of r

charArray[2] has a value of i

charArray[3] has a value of e

charArray[4] has a value of n

charArray[5] has a value of d

charArray[6] has a value of s

charArray[7] has a value of
```

```
My favourite comedy is Friends
```

Notice that each of the characters is enclosed with single quote marks - double quote marks are reserved for strings. The character and string format specifiers are also used (`%c` and `%s` respectively).

Caution: Use the right slash symbol for the null character.

1.9 Structure

Before a structure is used, it has to be declared. The syntax for structure is as follows:

```
struct  <structure name>
{
        Data_type     structure_element 1;
        Data_type     structure_element 2;
        Data_type     structure _element 3;
        ….
        ….
        Data_type      structure_element n;

};
```

Once, the new structure data type has been defined one or more variable can be declared of that structure type. Consider a structure declared as follows:

```
struct student {
        char    name[10];
        int     rollno;
        char    sex;  /*   m or f  */
        int     age;
        };
```

This defines a new data type called **student** to be a structure with the specified shape; name, rollno, sex and age. A variable of type struct student can be declared as follows :

```
struct  student collegestu, schoolstu;
```

This statement allocates space in memory and makes available space to hold structure elements. A structure can be declared in any of the format given below.

Format one:

```
struct student {
        char    name[10];
```

```
        int     rollno;
        char    sex;
        int     age;
    };

    struct  student collegestu,schoolstu;
```

Format two:

```
    struct
    {
        char    name[10];
        int     rollno;
        char    sex;            /*   m or f  */
        int     age;
    } collegestu, schoolstu;
```

Accessing and Initialization of Structure

Structure variable uses dot (.) operator to access structure element. Syntax to access the structure elements is

```
structure_name . structure_element_name
```

for example, To set the value to the structure element

```
collegestu.rollno = 1125;
collegestu.age = 21;
```

for example, To take the value from the structure element

```
int  rnumber = colledestu.rollno;
int  stuage = collegestu.age;
```

The following program illustrates how structure members can be accessed.

```
#include<stdio.h>
struct stud
{
  int rno;
  char name[20];
 };
main()
{
  struct stud s;
```

```
  printf("Enter the student's name:");
  scanf("%s",s.name);
  printf("Enter the student's roll number:");
  scanf("%d",&s.rno);
  printf("The student's details are:\n");
  printf("Name : %s\n",s.name);
  printf("Roll No. : %d\n",s.rno);
}
```

Output:

```
Enter the student's name:Adithya
Enter the student's roll number:100
The student's details are:
Name : Adithya
Roll No. : 100
```

The above program accesses the members of the structure by declaring a structure variable and accessing each member using a dot operator. In the above program, the structure members ***name*** and ***rno*** are accessed using the structure variable ***s*** and dot operator.

Array of Structure

Just like an array of variable, an array of structure can also be declared. Suppose a symbol table for 100 identifiers has to be made. The definitions can be extended like

```
char   id[100][10];
int    line[100];
char   type[100];
int    usage[100];
```

but a structure lets rearranging this spread-out information so that all the data identifier is collected into one lump:

```
struct {
    char   id[10];
    int    line;
    char   type;
    int    usage;
  } sym[100];
```

This makes **sym** an array of structures; each array element has the specified shape. Now, members can be referred to as

```
sym[i].usage++; /* increment usage of i-th identifier */
```

```
for( j=0; sym[i].id[j++] != '\0'; ) ...
  etc.
```

Thus, to print a list of all identifiers that has not been used, together with their line number,

```
for( i=0; i<nsym; i++ )
      if( sym[i].usage == 0 )
            printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Nested Structure

It is possible to declare a structure inside another structure. This means a structure can be made a member of another structure. The following program uses a structure called **date** as a member of another structure.

```
#include<stdio.h>
#include<conio.h>
struct date
{
 int dd;
 int mm;
 int yy;
};
struct stud
{
 char name[20];
 struct date d;
 int m1,m2,m3;
 float per;
 };
main()
{
  struct stud s;
  printf("Enter the Details of student\n");
  printf("\nName:");
  scanf("%s",s.name);
  printf("\nDate of Birth(dd mm yy):");
  scanf("%d%d%d",&s.d.dd,&s.d.mm,&s.d.yy);
  printf("\nMarks in 3 Subject:");
  scanf("%d%d%d",&s.m1,&s.m2,&s.m3);
  s.per=(s.m1+s.m2+s.m3)/3;
  printf("\nStudent Result");
  printf("\n%s has secured %f%%",s.name,s.per);
  getch();
 }
```

Output:

```
Enter the Details of student
Name:Ashwin
```

```
Date of Birth(dd mm yy):12 12 1980
Marks in 3 Subject:98
97
95

Student Result
Ashwin has secured 96.000000%
```

1.10 Pointers

A pointer in C refers to a variable that holds the address of another variable. In real time, pointers are a quite common way to get the contents of a variable. The unary operator `&' is used to return the address of a variable. Thus, the following sentence prints the address of a variable called ***a***.

printf("The address of the variable is: %u",&a);

If the address has to be assigned to a pointer variable, then an appropriate pointer variable has to be declared. A pointer variable is declared with an asterisk symbol preceding its name.

The following example illustrates this.

```
int a=5, *b, c;
b = &a;
c = *b;
```

b contains the address of a and `c = *b' means to use the value in b as an address, that is, as a pointer. The effect is that the contents of a is copied to c.

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array. Consider the following,

```
char *y;
char x[100];
```

y is of type pointing to character. The pointer variable y can be made to point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0]. This is legal and consistent.

Now `*y' gives x[0]. More importantly,

*(y+1) gives x[1]
*(y+i) gives x[i]

and the sequence

```
y = &x[0];
y++;
```

leaves y pointing at x[1]. In fact the name of the array is itself a pointer to the first element of the array.

The following program accepts elements for an array. The elements of the array are accessed using the name of the array.

```
#include<stdio.h>
main()
{
  int x,a[5];
  printf("Enter 5 elements\n");
  for(x=0;x<5;x++)
  {
    scanf("%d",&a[x]);
  }
  printf("The array elements are:\n");
  for(x=0;x<5;x++)
 {
    printf("%d\n",*(a+x));
 }
}
```

Output:

```
Enter 5 elements
1
2
3
4
5

The array elements are
1
2
3
4
5
```

Pointers and Structures

Consider a block of data containing different data types defined by means of a structure. For example, a personal file might contain structures, which look something like:

```
struct tag
{
    char lname[20];       /* last name */
    char fname[20];       /* first name */
    int age;              /* age */
};
```

Just like a variable of type structure, a pointer variable of type structure can also be declared. The following program declares a pointer variable of type struct tag and uses this pointer variable to access every member of the structure.

Example:

```
#include<stdio.h>
 struct tag
{
  char lname[20];
  char fname[20];
  int age;
}s;

      main()
      {
      struct tag *t
      t = &s;
      printf("\nEnter the first name.");

      scanf("%s",t-> fname);
      printf("\nEnter the last name:");
      scanf("%s",t-> lname);
      printf("\nAge:");
      scanf("%d",&t->age);
      printf("\nThe details are:");
      printf("Name:%s %s\nAge:%d", t-> fname, t-> lname,t->age);
      }
```

Output:

Enter the first name:Mahatma

```
Enter the last name:Gandhi

Age:60

The details are:Name:Mahatma Gandhi
Age:60
```

1.11 Summary

- C is a high-level programming language that forms the basis to other programming languages such as C++, PERL and Java. Dennis Ritchie developed it in 1972. He named it C because there was an existing programming language called B.
- In C, there are four data types: char, int, float and double. Each one has its own properties. For instance, they all have different sizes. The size of a variable can be pictured as the number of containers / memory slots that are required to store it.
- Variables are like containers in the computer's memory – one can store values in them and retrieve or modify them when necessary
- Control structure is used to control the sequence of the program flow. They are of two categories namely Selection and Loops
- A function can be thought of as a mini-program, where a group of statements are executed when the function is called.
- A function that calls itself is said to be a RECURSIVE function. Recursive function calls can be less efficient than a loop.
- An Array is a collection of variables of the same data type that are referenced by a common name. Each element in an array is associated with an index and occupies contiguous memory location.
- Using structure we can create a group of data of different data type, new structure data type has been defined one or more variable can be declared of that structure type.
- A pointer in C refers to a variable that holds the address of another variable. In real time, pointers are a quite common way to get the contents of a variable. The unary operator `&' is used to return the address of a variable.

1.12 Review Questions

1. Discuss the data types in C language.
2. Define variable, constant
3. Explain the operators available in C language
4. Discuss all the possible ‘if’ statements
5. Differentiate between ‘for’ , ‘while’ and ‘do-while’ loops
6. Explain function and its significance.
7. Discuss Arrays and its types, advantages

8. Explain structures.
9. What is the difference between a variable and a pointer variable.

Unit II

2.0 Objective

- Features and Basic concepts of OOPs
- Applications of OOP
- Classes and Objects
- Functions and its Types
- Function Overloading

2.1 Introduction to Object Oriented Programming

Until recently, programs were thought of as a series of procedures that acted upon a set of data. A procedure, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures and the trick in programming was to keep track of which functions called which other functions, and what was the data that was changed. To make sense of this potentially confusing situation, structured programming was created. The principle idea behind structured programming is simple: divide and conquer. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

For example, consider a complex task of computing the average salary of every employee of a company. It can, however, be broken down into subtasks such as, finding out what each person earns, counting the number of people, summing up all the salaries and dividing the total by the number of people. The total of all the salaries can also be broken down into subtasks such as, getting each employee's record, accessing the salary, adding the salary to the running total and getting the next employee's record. Obtaining each employee's record can, in turn, be broken down as opening the file of employees, going to the correct record and reading the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems. But, by the late 1980s, some of the deficiencies of structured programming had become too clear. There was a difficulty in separating the data structures from functions that manipulated the data. The other disadvantage was that the programmers found themselves constantly reinventing new solutions to old problems. The idea behind reusability is to build components that have known properties, and then to be able to plug them into the program as and when it is needed. This is modeled after the hardware world. For instance, when engineers need a new transistor, they does not invent one, they go to a big bin of transistors and finds one that works the way they need, or perhaps modify it. No similar option existed for a software engineer.

Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate the data. The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single "object", a self-contained entity with an identity and certain characteristics of its own.

Design is the process whereby the user can specify a method for solving a problem. It allows us to realize a solution in a manner that can be tested and verified. It is essentially a model of the final artifact that is to be created. Neither process is likely to be achieved without some effort and both will need iteration through a number of cycles

- Re-usability
- Extensibility
- Decomposability
- Composability
- Understandability
- Continuity
- Protection
- Problem domain
- Function to be performed
- Behavior
- Partitioning to expose detail
- Postpone complexity

Object model:

It is a graphical representation that describes the structure of objects within the system - identities, relationships to other objects, their attributes and operations. It provides the context within which the other two models are placed.

Dynamic model:

This model shows the behavior of the system over time, it shows how the system changes state in response to various events. State diagrams are used to describe the sequence of events. It is concerned with when an operation occurs rather than what operation is done.

Functional model:

It looks at how values change and is transformed. It is concerned with what the system does not how or when. It uses data flow diagrams to model the dependencies between values and the computations involved.

Requirements should be separated true requirements from design and implementation decisions. It should be a statement of needs and not a proposal for a solution. Performance specifications and demands for interaction with external systems are OK as the meeting of Software Engineering requirements such as modularity, testability and future extensibility.

The Object Model shows the static data structure of the real world model. It shows real world object classes and their relationship to each other. It is usually developed first because it is better defined than the dynamic and functional models. It is fewer dependants on implementation details. It acts as a good basis for communication as it is relatively easy to understand.

2.2 Features of OOPs

Real world Objects

When approaching a programming problem in object-oriented programming, the problem is divided into objects and not functions as in structured programming. Thinking in terms of objects has a helpful effect on how easily the programs can be defined. This is because of the close match between the objects in programming sense and objects in real world. Things that become objects in object-oriented programs are limited only to imaginations.

State and Abilities

The match between programming objects and real world objects is the result of combining data and member functions. This is a very important feature of object-oriented programming. Many real world objects have both a state (characteristics that can change) and ability (things they can do). In C++, an object's data corresponds to its state and its member function corresponds to its ability. Object-oriented programming combines the programming equivalent of states and abilities that are represented in a program by data and functions into a single entity called an **object**. The result is a programming entity that corresponds with many real-world objects.

Classes

In object-oriented programming, objects are instances of classes. For example, a data type **int** that refers to integer is predefined in C++. Any number of variables can be created of type **int** as it is needed in the program. In a similar way, many objects of the same class can be defined. A class thus serves as a plan or template. It specifies what data and what function will be included in the objects of the class. Defining the class does not create any objects, just as the existence of type **int** does not create any variables of type **int**. A class is thus a description of a number of similar objects. An object can be called an instance or an instantiation of a class because the object is an instance of the specifications provided by the class.

2.3 Basic Concepts of OOPs

C++ and Object-Oriented Programming

C++ fully supports object-oriented programming features like: encapsulation, inheritance, and polymorphism.

Principles Of OOPS

The following are the OOPS concept,

a. Data abstraction
b. Data Encapsulation
c. Inheritance
d. Polymorphism

a. Data Abstraction

A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction

b. Data Encapsulation

The property of being a self-contained unit is called encapsulation. With encapsulation, data hiding is possible. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally.

Just as a refrigerator is used without knowing how its compressor works, a well-designed object can be used without knowing about its internal data members.

C++ supports the properties of encapsulation through creation of user-defined types, called classes. Once created, well-defined class acts as a self contained entity. The internal working of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

c. Inheritance and Reuse

The workers in manufacturing company were presented with two options. They could either start from scratch to create a new model, or they can modify an existing model A that is nearly perfect, but needed some additional capabilities. The supervisor preferred not to start from the scratch. He decided to build a new model B from the existing one, A, adding additional capabilities and introduce it as a new model. Therefore, the new model B is nothing but the old model A but a specialized one with new features. This describes what inheritance is all about.

C++ supports inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The model B is derived from the model A and inherits all its qualities, but can be added to it as needed.

d. Polymorphism

Continuing forward the example used in inheritance, the model B and A might internally respond in a different manner, which is no use for the user to know. He uses it as prescribed and everything is smooth and fine.

C++ supports the idea that different objects perform everything in the correct manner through what is called polymorphism. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms.

2.4 Applications of OOP

A Simple Program

This section reviews this program in more detail. The following discussion demonstrates the constituents of a C++ Program

```
#include <iostream.h>
int main()
{
  cout << "Hello World!\n";
  return 0;
}
```

The **output** is:
Hello World!

In the first line, the file **iostream.h** is included into the current file. This is how it works: the first character is the # symbol, which is a signal to the preprocessor. Each time the compiler is started, the preprocessor is run. The preprocessor reads the source code, looking for lines that begin with the pound symbol (#) and acts on those lines before the compiler runs.

Include is a preprocessor instruction which says that what follows is a filename to be found. The angle brackets around the filename tell the preprocessor to look for this file. If the compiler is set up correctly, the angle brackets will cause the preprocessor to look for the file **iostream.h** in the directory that holds all the **.h** files in the compiler. The file **iostream.h** (input-output-stream) is used by **cout**, which assists in writing to the screen. The effect of the first line is to include the file **iostream.h** into this program. The preprocessor runs before each time the compiler is invoked. The preprocessor translates any line that begins with a pound symbol (#) into a special command, getting the code file ready for the compiler.

The actual program begins with a function named **main()**. Every C++ program has a **main()** function. A function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but **main ()** is special. When the program starts, **main ()** is called automatically.

Main(), like all functions, must state the kind of value it will return. The return value type for **main ()** in **HELLO.cpp** is int, which means that this function will return an integer to the operating system when it completes. In this case, it returns the integer value 0, as shown by the statement **return 0**. Some compilers allow declaring **main ()** to return **void**. Some operating systems enable testing the value returned by a program. The convention is to **return 0** to indicate that the program ended normally.

All functions begin with an opening brace (**{**) and end with a closing brace (**}**). Everything between the opening and closing braces is considered a part of the function. The main intention of this program is described within these two braces.

The object **cout** is used to print a message to the screen. The **cout** is followed by the output redirection operator (<<), which writes everything to the screen. To print a string, enclose them in double quotes (**"**).

The final two characters, **\n**, tell **cout** to put a new line after the words Hello World! The **main ()** function ends with the closing brace.

A Brief outlook at cout

To print a value to the screen, write the word cout, followed by the insertion operator (<<). The following example illustrates how this is used.

```
//Using cout
#include <iostream.h>
int main()
{
        cout << "Hello there.\n";
        cout << "Here is 5: " << 5 << "\n";
        cout << "The manipulator endl writes a new line to the screen.";
        cout << endl;
        cout << "Here is a big number:\t" << 70000 << endl;
        cout << "Here is the sum of 8 and 5:\t" << (8+5) << endl;
        cout << "Here is a fraction:\t\t" << (float) 5/8 << endl;
        cout << "And a very very big number:\t";
        cout << (double) 7000 * 7000 << endl;
        cout << "We are C++ programmers!\n";
        return 0;
}
```

Output:

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a big number:           70000
Here is the sum of 8 and 5:     13
Here is a fraction:             0.625
And a very very big number:     4.9e+07
We are C++ programmers!
```

The simplest use of cout, printing a string or series of characters namely **cout << "Hello there.\n";**. The symbol **\n** is a special formatting character. It tells **cout** to print a newline character to the screen.

Three values are passed to cout on the next line, and the insertion operator separates each value. The first value is the string **"Here is 5:"**. Next, the value 5 is passed to the insertion operator which in turn is followed by the new line character (always in double quotes or single quotes). This causes the line

Here is 5: 5

to be printed to the screen.

The next line causes the informative message

The manipulator endl writes a new line to the screen.

to be printed, and then the manipulator **endl** write a new line to the screen.

A new formatting character, \t, is introduced in the line **cout << "And a very very big number:\t";**. This inserts a tab character.

Line 9 shows that not only integers, but long integers as well, can be printed. The line **cout << "Here is the sum of 8 and 5:\t" << (8+5) << endl;** demonstrates that cout will do simple addition. The value of 8+5 is passed to cout, but 13 is printed.

On line **cout << "Here is a fraction:\t\t" << (float) 5/8 << endl;** the value 5/8 is inserted into cout. The term (float) tells **cout** that this value is to be evaluated as a decimal equivalent, and so a fraction is printed.

On the line **cout << (double) 7000 * 7000 <<**, the value 7000 * 7000 is given to **cout**, and the term (double) is used to tell **cout** that this is to be printed using scientific notation.

The line **cout << "We are C++ programmers!\n";** types the string **We are C++ programmers!**

Comments

When a program is written, it should always be clear and self-evident what is being done. Comments are used for this purpose. Comments are text that is ignored by the compiler, but that may inform the reader of what is being done at any particular point in the program.

Types of Comments

C++ comments come in two flavors: the double-slash (//) comment, and the slash-star (/*) comment. The double-slash comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows this comment, until the end of the line. The slash-star comment tells the compiler to ignore everything that follows until it finds a star-slash (*/) comment mark. These marks will be referred to as C-style comments. Every /* must be matched with a closing */.

Using Comments

Generally, program should have comments at the beginning, telling what the program does. Each function should also have comments explaining what the function does. In short, to write a good program, supplement it with comments to increase understandability. Comments that state the obvious are less than useful. In fact, comments can be counterproductive because the code may change and the programmer may neglect to update the comment. What is obvious to one person may be obscure to another, so judgment is required. The bottom line is that comments should not say what is happening, comments should say why it is happening.

2.5 Defining Classes and Objects

User defined data types can be defined using the **typedef** keyword, but creating a data type using this keyword can be inadequate in most real time situations. For example, a business application may find the storage and manipulation of dates a basic requirement. In such cases, the **typedef** would not help.

C++ offers the capability to define customized user defined data types like the one describe below:

```
struct date {
        int dd;
        int mm;
        int yy;
    }
```

In the above type, the biggest drawback lies in the weak ties it would have with its validations and related operations. For example, in case of the data type int, the user can enter any numerical values. This is because the validations for this data type have been tied to **int** alone and hence can accept any number while the maximum for month can be only 12. C++ provides a simple and elegant means of ensuring this – include the validations as a part of the date declaration itself. This could be done by using the **class** construct.

A class declaration in C++ is very much similar to the **struct** declaration as shown in the below example.

```
class date {
        int dd;
        int mm;
        int yy;
     }
```

Note that the keyword **struct** being replaced by the word class, however, this definition is not complete. The main idea behind using the classes is the binding of data

along with functionality. The modified declaration of the date class, along with a validation function would look like this.

```
class date  {
            int dd;
            int mm;
            int yy;
            void valid(int d, int m)
            {
                  if (d > 31)
                        cout << "\nDate not valid";
                  if (m > 12)
                        cout <<"\nMonth not valid";
            }
      }
```

The function "valid" is now a member of the **date** data type. Such functions are referred to as **member functions** or **methods**. The data members within the class – **dd, mm, yy** are in turn called **member data**.

Data abstraction

Data abstraction is the ability to create user-defined data types for modeling real world objects using built-in data types and a set of permitted operators. The construct that helps creating such data types and implement data abstraction in C++ is the class. It helps in the creation and usage of user-defined or **Abstract Data Types**.

Encapsulation

Encapsulation or data hiding is the mechanism that associates the code and the data that it manipulates into a single unit and keeps them safe from external interference and misuse. The class feature of C++ implements the concept of encapsulation by providing **access specifiers**. **Access specifiers** control the visibility status of the members of a class. There are three access specifiers in C++ - private, public and protected. The members of a class declared private are accessible only within the class i.e. only to other members of the same class. The members declared as public are accessible anywhere from within the class, or outside the class. The protected access specifier will be discussed in detail in the next chapter. The following example illustrates the usage of private and public access specifiers.

```
class date  {
            private:
                  int date;
                  int mm;
                  int yy;
            public:
```

```
        void validation(int d, int m)
        {
                if(d > 31)
                        cout <<"\nInvalid date";
                if(m> 12)
                        cout <<"\nInvalid month";
        }
};
```

Access specifiers serve the important purpose of defining the boundary between the accessible and inaccessible parts of a class. It is the privilege of the designer of the class to determine which parts of a class must be obscured from the user's point of view and which should not.

The design of a data type would be incomplete if the validity of the data stored within it is questionable. In the case of the class date, if the user was permitted to modify the values of **dd**, **mm** and **yy** directly, the possibility of illegal values being entered is relatively high. That is why the three variables **dd**, **mm** and **yy** have been hidden from the user's view in the above example. If the date has to be set, the appropriate functions can be invoked and the values set after making proper validations.

The member data and functions are accessed with the help of the member access operator ('.'). For example,

```
date date1;
date1.dd=10;            // not valid, since dd
                        is // a private member
date1.valid(10,12);     // perfectly valid
```

The variable date1, of type date, is called an object. An object is an instance of a class. If the object happens to be a pointer, the operator to be used is ('->'). For example, the same members could be accessed as follows:

```
date *date1;
date1->dd=10;           // not valid, since dd
                        is // a private member
date1->valid(10,12)     // perfectly valid
```

While designing a class the designer must examine each member of the class before declaring it as private or public. Most class definitions keep the data elements within the private section and the methods in the public section. In some cases, it might also be necessary to declare some methods as private. These methods will act as interface functions, accessible only by other methods within the class.

Dynamic Memory Manipulation

C++ provides two special operators to perform memory management dynamically. These are the **new** operator for the dynamic memory allocation and the **delete** operator for dynamic memory deallocation.

The new operator

The general format of the new operator is as follows:

DataType *variable_name = new DataType [size is integer];

where *variable_name is the pointer to that type which will be returned; new is the operator, DataType is again the type for which memory is to be allocated and size is the number of items to be allocated (this is optional).

Example:

```
int *d;
d = new int;
```

allocates memory for an integer and returns a pointer of type int, which is assigned to d.

The delete operator

The counterpart of **new** operator, **delete**, ensures the safe and efficient use of memory. This operator is used to free the block of memory allocated by the **new** operator. Although the memory allocated is returned to the system when the program terminates, it is safer to explicitly free memory using delete operator. The format of this operator is

delete pointervariable;

where **pointervariable** is the pointer returned through the new operator. The C++ statement

delete d;

If the new operator is used to explicitly allocate memory to a variable, the delete operator must be used to destroy it. Though the variable goes out of scope when the program gets terminated, the memory area allocated by the new operator is not released to the system unless the delete operator explicitly releases it.

2.6 Functions and its Types

A function is a single comprehensive unit that performs a specified task. This specified task is repeated each time the function is called. Functions break large computing tasks into smaller ones. Functions work together to accomplish the goal of a whole program. Moreover, functions increase the modularity of a program and reduce code redundancy.

Every program must contain one function named **main ()** where the program always begins execution. The main function may call other functions, which in turn, may

again call other functions. When a function is called, the control is transferred to the first statement in the called function. After the function is executed, control returns to the calling function and execution continues with the evaluation of the expression in which the call was made. A value can be returned when a function completes the specified task and that value can be used as an operand in an expression.

Functions can communicate data between themselves in two ways – one through global variables and the other through an argument or parameter list.

Functions in C++ can be classified into two types – library functions and user-defined functions. Library functions are a set of pre-defined functions, **printf()**, **fflush()**, **strcat()**, etc. The main distinction between these two categories is that the user need not write library function, whereas a user-defined function has to be developed by the users at the time of the writing the program. However, a user-defined function can later become a part of the C++ library.

Function prototype

In C++, all functions must be declared before being used. This is normally accomplished using a function prototype. When prototypes are used, the compiler finds and reports any illegal type conversions between the types of arguments used to call function and the type definition of its parameters.

The general form of a function prototype is :

type function-name (type parameter_name1, type parameter_name2,…type parameter_namen)

The use of parameter names is optional.

Function Definition

A function definition introduces a new user-defined function to the program by declaring the type of value it returns and its parameters and specifying the statements that are executed when the function is called.

```
#include<iostream.h>
void main()
{
        int x, y, z;
        int add(int x, int y);
        cout <<"\nEnter a number :";
        cin >> x;
        cout <<"\nEnter another number:";
        cin >> y;
        z=add(x,y);
```

```
        cout <<"\nThe result is : " << z;
}
int add(int x, int y)
{
        return (x + y) ;
}
```

The **output** of the program is:

```
Enter a number: 5
Enter another number: 7
The result is : 12
```

The function in which the function call is contained is known as the **calling function** and the function named in the call is said to be the **called function**. When the function is required to return a value, the return type has to be specified explicitly. The general form of a C++ function is given below:

```
return type_functionname(argument list)

{
        variable declarations;
        function statements;
        return value;
}
```

Passing values to functions

Arguments to a function are usually passed in two ways. The first is termed as **call by value**. In call by value, a copy of the variable is made and passed to the function as argument. With this method, changes made to the parameters of the function have no effect on the variables in the calling function because the changes are made only to the copies. On the other hand, call by reference is a method in which the address of each argument is passed to the function. By this method, the changes made to the parameters of the function will affect the variables in the calling function. An example to demonstrate **call by reference** and **call by value** is shown below.

```
//call by value
#include<iostream.h>
void main()
{
        int a=1, b=2;
        void swap(int, int);

        swap(a,b);
        cout <<"\na=" << a << "b=" << b;
}
```

```
void swap(int a, int b)

{
        int temp;
        temp=a;
        a=b;
        b=temp;
}
```

The **output** will be:

a=1b=2

Note: The values of the variables are not exchanged

```
//call by reference
#include <iostream.h>
void main()
{
        int a=1, b=2;
        void swap(int *, int *);
        swap(&a, &b);
        cout << "\na=" << a << " b=" << b;
}

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

The **output** will be:

a=2 b=1

Note : The values of the variables are exchanged

In the first program, swap is done only on the copies of the variables **a** and **b**. But in the call by reference version swap is done on the original variables themselves because the address of the variables is passed down to the function.

Recursive Function

C++ supports a feature called **recursive function**. A **recursive function** is a function that makes a call to itself. **Recursive** is the process of defining something in terms of itself, and is sometimes called circular definition. Recursive functions can be effectively used in applications in which the solution to a problem can be expressed by successively applying the same solution to the subsets of the problem. A common example of recursion is the evaluation of the factorial of a given number.

```
#include<iostream.h>
void main()
{
        int j;
        int factorial(int);
        for(j=0; j < 10; j++)
                cout << '\n' << factorial(j);
}
```

```
factorial (int n)
{
        int fact;
        if(n==0)
                return 1;
        else
        {
                fact = n * factorial(n-1);
                return fact;
        }
}
```

The **output** is :

```
1
1
2
6
24
120
720
5040
40320
362880
```

Inline functions

In any programming language, the best way to increase the modularity of a large program is by using functions. The proper usage of function to perform independent tasks greatly increases the readability of a program and reduces maintenance costs. However, there is one hitch to this concept. A call to the function causes the program to be suspended and control to be passed to the function being called. The program branches to the called function, and when the execution is complete, control is returned to the program and execution continues from the line following the function. Thus, a function call involves an overhead in terms of the time taken to execute these additional steps.

Sometimes, a member function may be very small (say, just a single line of code, like the **getword** function in the word class). In such cases, the time taken to make the function call is far more than that taken to execute the function code itself. Hence, replacing the function call with the actual body of the function itself can make the program faster. This can be made possible in C++ by declaring the function as inline. A function can be made inline in two ways:

- By writing the code of the function itself within the class declaration, i.e., without using the scope resolution operator.
- By using the inline keyword before the function definition. For example,

```
inline char * word::getword()
{
    return this ->str_word;
}
```

The inline specifier is only a suggestion to the compiler. All functions declared, as inline need not be treated as inline by the compiler. If the function happens to be recursive or if the function body is too large, the compiler just ignores the inline specification. This is because it will not be a good idea to substitute 300 lines of function code in place of a simple function call. In case of recursive functions, the function body cannot be substituted each time the function is called because the number of recursions is not known at the time of compilation.

2.7 Passing Objects as Functions Arguments

Here is an example for passing an object as an argument ,

```
#include <iostream.h>
struct Disp
{
        int x;
        int y;
};

void get(Disp dd)
{
```

```
   cout << "Displaying X and Y";
   cout << "\nX is      = " << dd.x;
   cout << "\nY is     = " << dd.y;
}

int main(int argc, char* argv[])
{
   int a,b;

   cout << "Values of X and Y are\n";
   cout << "Width: ";
   cin >> a;
   cout << "Height: ";
   cin >> b;

   Disp d1 = {a, b};
   get(d1);

   getchar();
   return 0;
}
```

The structure name is "Disp" and it has two objects x and y. we cannot access these variables directly from outside the scope of the structure. But can access those variables by creating an object for the structure. Generally accessed in the following syntax,

objectname.variable ;

in the above program accessed as ,

dd.x;
dd.y;

and the object is passed as an argument to the function from the main function.

2.8 Function Overloading

The origin of the word polymorphism is simple. It is derived from two Greek words **poly** (many) and **morphos** (form) – multiform. Function and operator overloading are crucial elements in C++ programming. Not only do these two features provide most support for compile time polymorphism, it also adds flexibility and extensibility to the language. Function overloading is simply the process of using the same name for two or more functions. Although operator overloading is similar to function overloading, it is better to know about function overloading before going in for operator overloading.

Overloading

Overloading grants flexibility to C++. Overloading lets the programmer use code with less effort, because it extends operations that are conceptually similar in nature. Consider an example function, **Display** which takes an argument and displays it on the screen.

- Display("Display on the screen");
- Display(123456789);
- Display(98.7654321);

The compiler may not think that a character string is similar to a floating-point number, but a programmer certainly thinks that the **Display** function for strings, integers and floats work in a similar way. Thus, the emphasis is on user friendliness. The advantage obtained by overloading does not come without a price, though, because this advantage indicates added complexity to the compiler. The compiler overcomes this by providing each function a unique signature and calling the appropriate function at run time. The signature of a function includes the type, number and sequence of parameters.

Overloading **can also be done for operators like unary operators such as ++, --) and binary operators such as +, +=, <=, etc. The concept of overloading is called** static binding **or** static polymorphism**.**

Function Overloading

Functions that share the same name are said to be overloaded and the process is referred to as **Function Overloading**. Passing different types of parameters or different number of parameters to the function incorporates this. It is only through the signature that the compiler decides which function to call in a given situation.

The minimum requirement for function overloading is that, the function must differ in type, number or sequence of parameters. Two functions of the same name cannot be overloaded if their return types alone are different.

Non Member Overloaded Function

Any function can be overloaded in C++. The following is an example of code using overloaded **FunctionName()** all contained in the same file, and all having file scope.

```
#include<iostream.h>

int FunctionName(int);
double FunctionName(double i);
main()
```

```
{
        cout << FunctionName(10) << endl;
cout << FunctionName(10.33) << endl;
return 0;
}

int FunctionName(int i)
{
i+=10;
cout << "Inside int FunctionName(int i)" << endl;
return 0;
}
double  FunctionName (double i)
{
        i+=10.5;
        cout << "Inside double FunctionName(double i)" << endl;
return i;
}
```

The **output** of the program would be:

```
Inside int FunctionName(int i)
0
Inside double FunctionName(double i)
20.83
```

Overloaded Member Function

The primary use of overloading is with class member function. When more than one member function with the same name is declared in a class, the function is said to be overloaded in that class. The scope of the overloaded function is restricted to a particular class. Other classes might use the same name again, overloading it differently or even not overloading it at all. Consider the following code:

```
#include<iostream.h>
class Sample
{
        private:
                int value;


public:
                void function(int v)
                {
                        value=v;
                }
```

```
        int function()
        {
                return value;
        }
};

void main()
{
        Sample s;
        s.function(7);
        int i=s.function();
        cout << i <<endl;
}
```

The **output** of the above code would be:
7

The above code illustrates two overloaded functions, one function to read and the other one to write a variable. Overloaded functions need to differ in either or both of the following ways:

- The functions must contain different argument types.
- The functions must contain a different number of arguments.
- At least one of the arguments must be different.
- If the type of arguments is the same, at least the sequence must be different.

The return type of a function is not a factor in distinguishing overloaded functions.

2.9 Summary

- Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate the data.
- In object-oriented programming, objects are instances of classes. A class thus serves as a plan or template. It specifies what data and what function will be included in the objects of the class.
- A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction
- The property of being a self-contained unit is called encapsulation. With encapsulation, data hiding is possible. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally.

- C++ supports inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The model B is derived from the model A and inherits all its qualities, but can be added to it as needed.
- C++ supports the idea that different objects perform everything in the correct manner through what is called polymorphism. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms.
- The actual program begins with a function named **main()**. Every C++ program has a **main()** function. A function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but **main ()** is special. When the program starts, **main ()** is called automatically.
- Data abstraction is the ability to create user-defined data types for modeling real world objects using built-in data types and a set of permitted operators.
- **Access specifiers** control the visibility status of the members of a class. There are three access specifiers in C++ - private, public and protected.
- The counterpart of **new** operator, **delete**, ensures the safe and efficient use of memory. This operator is used to free the block of memory allocated by the **new** operator.
- Functions in C++ can be classified into two types – library functions and user-defined functions. Library functions are a set of pre-defined functions, **printf()**, **fflush()**, **strcat()**, etc. The main distinction between these two categories is that the user need not write library function, whereas a user-defined function has to be developed by the users at the time of the writing the program
- **Recursive** is the process of defining something in terms of itself, and is sometimes called circular definition. Recursive functions can be effectively used in applications in which the solution to a problem can be expressed by successively applying the same solution to the subsets of the problem

2.10 Review Questions

1. Features and basics of OOPS
2. Discuss classes and objects
3. What is Dynamic memory manipulation?
4. What is the significance of new and delete operator
5. Define 1) Function 2) Function Prototype 3) Recursive function 4) inline function
6. Explain Function overloading
7. What are the advantages of using a function?

Unit III

3.0 Objective

- Constructors and Destructors
- Constructor Overloading
- Dynamic initialization of objects
- Copy constructor
- Friend Function
- Operator Overloading
- Type Conversions

3.1 Constructors

A ***constructor*** is a special function, which is having the same name as that of the class. The constructor will be called automatically when a new instance of the class is created. The constructor also differs from the function in that it has no return type.

To make obvious the concept of constructor program is as follows:

```
#include<iostream.h>
class Rectangle
{
    int x, y;
  public:
    Rectangle();
    void area ();
  private:
    int z;
  };
Rectangle::Rectangle( )
{
  cout<< "Enter x and y values  "<<endl ;
  cin>>x;
  cin>>y;
}
void Rectangle::area()
{
  z=x*y;
  cout<< "Area : " << z;
}
main()
```

```
{
  Rectangle r;
   r.area();
}
```

Output:

```
Enter x and y values
2
3
Area: 6
```

In this program, there is no separate function for accepting the value for the instance variables; a constructor is used instead. Hence, when the object is created as shown in the first statement of the ***main()***, the constructor is called automatically. The programmer need not worry about ensuring if the functions are called in proper order. The constructor then accepts the value for the instance variables and the final output is got when the function ***area()*** is called.

Difference between Constructor and member function

Constructor	Member function
Constructor has the same name as that of the class.	The name of the function can be different.
Return type is not allowed	Functions have return type.
Constructor is called automatically when an object is created.	Functions can be invoked only through objects.

3.2 Constructor Overloading

Just like a function can be overloaded, the same way constructor can also be overloaded. This means that a class can have more than one constructor but the number of arguments or the data type of the arguments passed to them differs.

```
#include<iostream.h>
class Poly
{
 int a,b,c;
 public:
  Poly()
  {
    a=0;
    b=0;
    c=0;
    cout<< " Not a polygon"<<endl;
```

```
    }
   Poly(int x)
   {
     a=x;
     cout<< " It is a square of side "<< a<<endl;
     cout<< " Perimeter of the Square :"<< 4*a<<endl;
    }
   Poly(int x, int y)
   {
     a=x;
     b=y;
     cout<< " It is a rectangle of height "<< a<< "and width "<< b<< endl;
     cout<< " Perimeter of the Rectangle :"<< 2*(a+b)<<endl;
    }
   Poly(int x, int y, int z)
   {
     a=x;
     b=y;
     c=z;
     cout<< " It is a Triangle "<< endl;
     cout<< " Perimeter of the Triangle :"<< (a + b + c)<<endl;
    }

};

main()
{
Poly p1,p2(5),p3(5,3),p4(1,2,3);
}
```

Output:

```
Not a polygon
It is a square of side 5
Perimeter of the Square :20
It is a rectangle of height 5and width 3
Perimeter of the Rectangle :16
It is a Triangle
Perimeter of the Triangle :6
```

The class Poly has four types of constructors. One of it uses no argument while the others use one, two and three arguments. Depending on the requirement, the required constructor is called by passing the corresponding number of arguments.

3.3 Destructors

The Destructor can be considered to be the opposite of constructor functionality wise. It is automatically called when an object is released from the memory, either because it goes out of scope or because it is an object dynamically assigned and is

released using operator delete. The destructor must have the same name as the class with a tilde (~) as prefix and it must return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed the memory that it has used has to be released.

The following program uses a destructor. The destructor is called automatically when the program terminates.

```
#include<iostream.h>
class Square
{
  int a;
  public:
     Square();
   ~Square();
     int peri();
};
Square::Square()
{
 cout<< "Enter the length of the side of a square "<<endl;
 cin>>a;
}
int Square::peri()
{
  int p;
  p=4*a;
  cout<< " Perimeter of the square is :   " ;
  return p;
}
Square::~Square()
{
   cout<< " \n  Destructor called :End of program";
}
main()
{
  Square s;
  cout<<s.peri();
 }
```

Output:

```
Enter the length of the side of a square
5
 Perimeter of the square is :   20
```

The above program uses only two statements in the main. The first statement in the main() creates an object of class Square thereby calling the constructor. This is then

followed by a call to the member function ***peri()*** to compute the perimeter of the square. The program then ends calling the destructor.

Difference between Constructor and Destructor

Constructor	Destructor
Constructor has the same name as that of the class.	It has the same name as that of the class but the name is preceded by a tilde (~) symbol
Constructor is called automatically when an object is created.	Destructor is called automatically when the object goes out of scope.

3.4 Dynamic initialization of objects

Class objects can be initialized dynamically too. That is, the initial values for the object's data member may be provided during run time. One advantage of dynamic initialization is that various initialization formats can be provided, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long-term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. The Program below illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
#include<iostream.h>
class Fixed_deposit
{
        long int P_amount;     //Principal amount
        int Years;             //Period of investment
        float Rate;            //Interest rate;
        float R_value; //Return value of amount
    public:
        Fixed_deposit(){}
        Fixed_deposit(long int p, int y, float r);
        Fixed_deposit(long int p, int y);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y)
{
        float r=0.21;
        P_amount=p;
        Years =y;
```

```
        Rate =r;
        R_value=P_amount;
        for(int i=1;i<=y; i++)
        R_value =R_value * (1.0 +r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount=p;
        Years =y;
        Rate =r;
        R_value=P_amount;
        for(int i=1;i<=y; i++)
            R_value =R_value * (1.0 +float (r)/100);
}
void Fixed_deposit ::display (void)
{
        cout << "\n";
        cout << "Principal Amount =" << P_amount << "\n";
        cout << "Return Value =" << R_value << "\n";
}
main()
{
        Fixed_deposit FD1, FD2,FD3; //deposit created
        long int p;     //principal amount
        int y;          //investment period, years
        float r; //interest rate, decimal form
        int R;          //interest rate, percent form

        cout << "Enter amount, period, interest rate(in percent)" << "\n";
        cin >> p >> y >>R;
        FD1=Fixed_deposit(p,y,R);

        cout << "Enter amount, period, interest rate(decimal percent)" << "\n";
        cin >> p >> y >>r;
        FD2   =Fixed_deposit(p,y,r);

        cout << "Enter amount and period" << "\n";
        cin >> p >> y;

        FD3   =Fixed_deposit(p,y);

        cout << "\n Deposit 1";
        FD1.display();

        cout << "\n Deposit 2";
        FD2.display();
```

```
        cout << "\n Deposit 3";
        FD3.display();
}
```

Output:

```
Enter amount,period,interest rate(in percent)
5000 5 15
Enter amount,period,interest rate(in decimal form)
5000 5 0.15
Enter amount and period
5000 5
Deposit 1
Principal Amount =5000
Return Value=10056.8

Deposit 2
Principal Amount =5000
Return Value=5037.61

Deposit 3
Principal Amount =5000
Return Value=12968.7
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms(1) and (2), and the third is invoked for the form(3). Note that, for form(3),the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for r.

3.5 Copy Constructor

Copy constructor is a special type of constructor in C++. The functionality of copy constructor is similar to copying variables.

Copying variables

For Example,

```
int a = 10;
int b=20;
```

With basic data types copying means creating a new variable of the same type and giving it the same value. The above statement does not mean assigning a value to an existing variable, they mean initializing variable. A different notation can be used for copying one variable to another.

For Example,
instead of int a=b;

This type of notation is sometimes called as **functional notation.** Anyhow to avoid confusion the equal sign can be used in between.

In the similar way objects can also be copied. Functional notation is widely used in **Copy Constructors.** Copy constructors provide another way to create objects, by making a copy of an existing object. They take a unique argument type.

Example

```
#include<iostream.h>
class CopyCon
{
  private:
        int intvar;
  public:
      CopyCon()
       {
              intvar = 10;
       }

      CopyCon(int v)
       {
                intvar=v;
       }

       CopyCon(CopyCon &obj)
       {
          intvar = obj.intvar;
          cout << "I am a copy constructor \n" ;
          cout << "intvar value inside copy constructor is " << intvar << endl;
       }
       void disp()
       {
         cout << "Value of intvar is " << intvar << endl;
       }
};

void main()
```

```
{
  CopyCon cobj;
  CopyCon cobj1(27);
  CopyCon cobj2 = cobj1;
  CopyCon cobj3(cobj1);

  cobj.disp();
  cobj1.disp();
  cobj2.disp();
  cobj3.disp();

  }
```

Output:

```
I am a copy constructor
intvar value inside copy constructor is 27
Iam a copy constructor
intvar value inside copy constructor is 27
Value of intvar is 10
Value of intvar is 27
Value of intvar is 27
Value of intvar is 27
```

3.6 Friend Function

So far, it has been stated that the private members of a class are not accessible anywhere outside the class. But these private members of a class are accessible to a function declared to be a friend of that class. Thus a friend function of a class is that function that has access to even the private members of the class. A function is declared a friend function by using the keyword ***friend***.

```
#include<iostream.h>
class Rectangle
{
   int width, height;
 public:
   Rectangle();
   int area () ;
   friend void perimeter ();
};

int Rectangle::area ()
{
    return (width * height);
}
```

```
Rectangle::Rectangle ()
{
 cout<<"Enter the width and height values:" <<endl;
 cin>>width;
 cin>>height;
}

void perimeter ()
{
 Rectangle r;
 int p;
 p=2*(r.width + r.height);
 cout<< "The perimeter of Rectangle is: "<< p;
}
main ()
{
 perimeter();

}
```

Output:

```
Enter the width and height values:
5
4
The perimeter of Rectangle is: 18
```

The program above illustrates the use of a friend function. In this case, the function ***perimeter()*** is a friend function of the class Rectangle and hence can gain access to the private members of the class Rectangle.

3.7 Operator Overloading

C++ has an important concept of Operator Overloading. Operator Overloading is a special case of function overloading, which allows assigning additional meanings to most of the standard C++ operators. When overloading operators, it is a good practice to ensure that the overloaded operator has a similar behaviour to the original operator. For example, it would make more sense to use the + operator for concatenation of strings than the = operator. Overloading operators does not change the precedence and associativity of the operator. New operators cannot be introduced using operator overloading.

An operator is overloaded using the keyword ***operator***. The general syntax for overloading an operator is as follows:

```
return_type operator operatorsymbol(argument)
{
```

```
  statement;
}
```

where,

return_type: refers to the data type of the variable or object to which the call to the overloaded operator is assigned. If the overloaded operator is not assigned to anything its return type is void.

operator is a keyword.

operatorsymbol refers to the operator that is to be overloaded.

argument refers to the variable or object that is used along with the operator. There should be a single argument in case of overloading a binary operator and there should be no argument when overloading unary operator.

An overloaded operator is called by giving the name of the object followed by the overloaded operator as follows:

Object_name overloaded_operator

Unary Operator Overloading

The following example shows how a unary operator is overloaded. The program uses the increment operator '++'. This operator requires only a single operand and hence takes no argument when it is overloaded.

```
#include<iostream.h>
class Uno
{
int a;
public:
Uno()
{
 a=1;
}
void operator ++()
{
  a++;
}
int display()
{
  return a;
}
};
```

```
main()
{
 Uno u;
 cout<< " Initial Value of a  : " <<u.display()<<endl;
 ++u;
 cout<< " Final Value of a  : " <<u.display();
}
```

Output:
Initial Value of a :1
Final Value of a :2

The class Uno has a single instance variable "a". Inside the overloaded operator function, the value of the instance variable is incremented. Hence when the function ++ is called using the object ***u***, the instance variable ***a*** of object ***u*** is incremented and the incremented value of ***a*** is then displayed when the function ***display()*** is called.

Binary Operator Overloading

The following program shows overloading of **binary operator** '+'. This operator takes a single argument.

```
#include<iostream.h>
class Bi
{
int x,y;
public:
Bi()
{
 x=1;
 y=2;
}
Bi operator +(Bi    b1)
{
Bi b2;
b2.x=x+b1.x;
b2.y=y+b1.y;
return b2;
}
void display()
{
 cout << " x:"<<x<<endl<< " y:" <<y<<endl;
}
};
main()
{
```

```
        Bi u,v,z;
        cout<< " Initial Values:- ";
        u.display();
        v.display();
        z.display();
        u=v+z;
        cout<< " Final Values:-  " ;
        u.display();
        v.display();
        z.display();
      }
```

Output:

```
Initial Values:-  x:1
y:2
x:1
y:2
x:1
y:2
Final Values:-   x:2
y:4
x:1
y:2
x:1
y:2
```

This program creates three objects of class *Bi.* Each time the constructor is called and the instance variables x and y of each of the object are assigned a value of 1 and 2 respectively. This is evident when the function ***display()*** of the class Bi is called using each of these objects. Note the statement,

```
u=v+z;
```

Here, the object ***v*** calls the overloaded operator **+**. Hence the variables ***x*** and ***y*** referred inside the function denotes the instance variable of the object ***v***. The statement also shows that an object ***z*** is also used along with the overloaded operator '+' on the right hand side. This object is passed as an argument to the overloaded operator function '+'. Thus the values of the instance variable of the object ***z*** are copied to the instance variable of object ***b1***. Thus the value of ***b1.x*** is same as that of ***z.x*** and the value of ***b1.y*** is same as that of ***z.y***.

Inside the overloaded operator function another object of class ***Bi*** called ***b2*** is created. The instance variable ***x*** of the object ***b2***, that is, ***b2.x*** is assigned the sum of ***v.x*** and ***b1.x*** (same as ***z.x***). Similarly the instance variable ***y*** of the object ***b2***, that is, ***b2.y*** is assigned the sum of ***v.y*** and ***b1.y*** (same as ***z.y***). The object ***b2*** is finally returned from the

function and hence the value of instance variables of this object are assigned to the corresponding instance variable of the object ***u***.

General Rules for Overloading Operators

There are certain restrictions and limitations to be kept in mind while overloading operators. The rules are:

- Only existing operators can be overloaded. New operators cannot be created.
- It is recommended not to change the basic meaning of an operator. That is, the plus (+) operator should not be redefined to subtract one value from another.
- Overloaded operators follow syntax rules of the original operators that cannot be overridden.
- Friend functions cannot be used to overload certain operators like =, (), [] and ->.
- Unary operators, overloaded by means of a member function take no explicit arguments. But those overloaded by a friend function take one reference argument.
- Binary operator overloaded through a member function takes one explicit argument and those that are overloaded through a friend function take two arguments.
- Binary operators such as +, -, * and / must explicitly return a value.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

Operator Overloading Restriction

There are some restrictions that apply to operator overloading.

- Operator functions cannot have default arguments.
- The operators that cannot be overloaded are:
  - **.**
  - **::**
  - **.**
  - *****
  - **?:**
  - **sizeof**
- One cannot alter the precedence of an operator.
- One cannot change the number of operands that an operator takes.

3.8 Type Conversions

Type conversion means converting a value in one data type to another. This is necessary so that a value of one data type can be conveniently assigned to a variable of a different data type.

Argument type conversion

During the function overload resolution, the compiler identifies and ranks the conversions that can apply to each argument in a function call to convert it to the type of

the corresponding parameter in each viable function. There are three possible outcomes of this ranking:

1. An exact match. The argument matches the type of the function parameter exactly.

2. Match with a type conversion. The argument does not directly match the type of the parameter but can be converted to the destination type.

3. No match. The argument cannot be made to match a parameter of the declared functions, because no type conversions exist between the argument and the corresponding function parameter.

When selecting the best viable function for a function call, the compiler selects the function for which the type conversions on the arguments are the "best". Type conversions are ranked as follows: an exact match is better than a promotion, a promotion is better than a standard conversion, and a standard conversion, and a standard conversion is better than a user-defined conversion.

Promotion Conversion

A promotion conversion is one of the following:

- An argument of type **char**, **unsigned char**, or **short** is promoted to type **int**. An argument of the type unassigned short is promoted to type **int** if the machine size of an **int** is larger than that of shorter integer otherwise, it is promoted to type unsigned **int**.
- An argument of type **float** is promoted to type **double**.
- An argument of an enumeration type is promoted to the first of the following type that can represent all the values of the enumeration constants: **int**, **unsigned int** or **unsigned long**.
- An argument of type **bool** is promoted to type **int**.

Standard Conversion

There are five kinds of conversions grouped in the category of standard conversions:

1. The integral conversions: the conversions from any integral type or enumeration type to any oher integral type.
2. The floating point conversions: the conversions from any floating point type to any other floating point type.
3. The floating integral conversions: the conversions from any floating point to any integral type or vice versa.
4. The pointer conversions: the conversion of a pointer of any type to the type **void**.
5. The **bool** conversions: the conversions from any integral type, floating point type, enumeration type, pointer type to the type **bool**.

User Defined Conversion (Casting operators &Conversion functions)

When the value of one object is assigned to another of the same type, such as

dist3 = dist1 + dist2;

the values of all the member data items are simply copied into the new object. The compiler knows how to do this automatically. Data conversion occurs when one type of data is assigned to another of different type. The conversion can be implicit such as

intvar = floatvar;

Casting provides explicit conversion:

intvar = static_cast<int>(floatvar);

For conversion of user-defined types, the programmers must define the conversion routines. Data conversion may seem unnecessarily complex or even dangerous.

However, the flexibility provided by allowing conversions outweighs the dangers.

Conversions between Objects and Basic Types

The following example performs the data conversion between the objects of class distance and float. The conversion is meant to convert Distance into meter and vice versa.

```
# include <iostream.h>
const float MTF = 3.280833F;
class Distance
{
        private:
                int feet;
                float inches;
        public:

                Distance()
                {
                        feet=0;
                        inches=0.0;
                }
                Distance(float meters)
                {
                        float fltfeet = MTF * meters;
                        feet = int(fltfeet);
```

```
                    inches = 12*(fltfeet-feet);
                }
                Distance(int ft, float in)
                {
                  feet =ft;
                  inches = in;
                }
                void getdist()
                {
                        cout << "\nEnter feet: "; cin >> feet;
                        cout << "Enter inches: "; cin >> inches;
                }
                void showdist() const //display distance
                {
                  cout << feet << "\'-" << inches << \'" ;
                }
                operator float() const //conversion operator
                {
                        float fracfeet = inches/12;
                        fracfeet += float(feet);
                        return fracfeet/MTF;
                }
};
void main()
{
        float mtrs;
        Distance dist1 = 2.35F;
        cout << "\ndist1 = "; dist1.showdist();
        dist1 = 1.00;

        cout << "\ndist1 = "; dist1.showdist();
        Distance dist2(5, 10.25);
        mtrs = float(dist2);
        cout << "\ndist2 = " << mtrs << " meters\n";
        mtrs = dist1;
        cout << " \n dist1 = " << mtrs << "meters" ;
}
```

Output:

```
dist1 = 7'-8.51949"
dist1 = 3'-3.37"
dist2 = 1.78435 meters
dist1 = 1meters
```

3.9 Summary

- A ***constructor*** is a special function, which is having the same name as that of the class.
- The Destructor can be automatically called when an object is released from the memory.
- Advantage of dynamic initialization is that one can provide various initialization formats, using overloaded constructors.
- A friend function of a class is that function that has access to even the private members of the class.
- Operator Overloading is a special case of function overloading, which allows assigning additional meanings to most of the standard C++ operators.
- An operator is overloaded using the keyword ***operator.***
- The operators that cannot be overloaded are: **. ,::, .*,?: and sizeof**.
- Type conversion means converting a value in one data type to another.

3.10 Review Questions

1. Describe the importance of constructor.
2. How do we invoke a constructor and destructor function?
3. Constructors do not return any values (True/False).
4. Explain copy constructor with an example.
5. What is friend function?
6. Define operator overloading.
7. Which operator cannot be overloaded?
8. What is the need for overloading an operator?
9. Explain type conversion with an example.

Unit IV

4.0 Objective

- Types of inheritance
- Virtual Base Class
- Pointers
- Pointers to Objects
- this Pointer
- Pointers to Base and the Derived Classes
- Virtual Functions
- Abstract classes

4.1 Introduction

One of the most useful features of classes is inheritance. It is possible to declare a class, which inherits the properties of another class or classes. This means that, with good class design, applications, which are based on proven re-usable code, can be built. Thus classes extend the feature of reusability of code, particularly complex code.

Definition: ***Inheritance*** is the mechanism by which a new class inherits the members of an already existing class.

Suppose a class A inherits the properties of a class B, it means "A inherits from B". Objects of class A thus have access to attributes and methods of class B without the need to redefine them.

If class A inherits from class B, then B is called ***superclass*** of A. A is called ***subclass*** of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass. In addition to the members inherited from the superclass, the subclass can also include additional members. Superclasses are also called ***parent classes***. Subclasses may also be called ***child classes*** or just ***derived classes***.

The subclass can again be inherited, making this class the superclass of the new subclass. This leads to a hierarchy of superclass/subclass relationships. The general format for inheriting a class is:

```
class new_class: access specifier existing_class
{
 members;
```

```
};
```

In this case,

new_class is the name of the subclass that is to be created.

access specifier can be either public, private or protected.

existing_class is the name of the superclass whose members are to be inherited by the subclass.
members refer to the extra instance variables and functions that the subclass can include in addition to the members of its superclass.

If a subclass inherits from more than one class then these classes along with an access specifier have to be separated by a comma ‘,’.

```
class new_class: access specifier existing_class1, access specifier existing_class2
{
  members;
};
```

where,

existing_class1 and *existing_class2* are the already existing class that are to be inherited.

Example

Consider a class called Sphere defined as follows:

```
class Sphere
{
        int r;
public:
        Sphere();
        int getDiameter();
        float getVolume();
};
Sphere::Sphere()
{
          cout<< “Enter radius”;
          cin>>r;
}
int Sphere:: getDiameter()
{
          return (r+r);
}
float Sphere:: getVolume()
{
          return (4/3)*3.14*r*r*r;
```

```
}
```

Now to create a class for ball and since a "ball is a sphere" the members of the sphere class can be inherited in creating the ball class. This way repeating the same set of coding can be avoided.

The ball class is created as follows:

```
class ball: public Sphere
{
public:
      void ballcolor();
};
void ball::ballcolor()
{
          cout<< " Ball color is red";
}
```

The class "ball" has all the member functions present in Sphere. In addition to the members ***r***, ***getDiameter()***, ***getVolume()*** inherited from the Sphere, an additional member function has been added ("***ballcolor()***") to hold the color of the ball.

When an object of class ball is created in the main(), all the functions (***getDiameter()***, ***getVolume()*** and ***ballcolor()***) can be called using this object. However reverse is not true. This means that using an object of class Sphere the function ***ballcolor()*** cannot be called.

```
#include<iostream.h>
class Sphere
{
        int r;
public:
                Sphere();
                int getDiameter();
               float getVolume();
};
Sphere::Sphere()
{
       cout<< "Enter radius:";
       cin>>r;
}
int Sphere:: getDiameter()
{
       return (r+r);
}
float Sphere:: getVolume()
{
       return (4/3)*3.14*r*r*r;
```

```
}
class ball: public Sphere
{
public:
      void ballcolor();
};
void ball::ballcolor()
{
  cout<< " Ball color is red";
}
main()
{
   ball b;
   cout<< b.getDiameter()<<"\n";
   cout<<b.getVolume()<<"\n";
   b.ballcolor();
}
```

Output:

```
Enter radius: 12
24
5425.92
 Ball color is red
```

When the object ***b*** of class ***ball*** is created the constructor of the super class is called. If there exists a constructor for ball class, first the constructor of Sphere class is called and then the constructor of the ball class is called. Through the constructor a value is assigned to the instance variable ***r*** and then the diameter, volume and color are computed.

Kinds of Inheritance

C++ provides three forms of inheritance:

- ***private*** inheritance (the default)
- ***protected*** inheritance
- ***public*** inheritance.

a. Private Inheritance

General form

```
class Derived:Base_Class
{
      access specifier:
            members;
};
```

```
or
class Derived: private Base_Class
{
        access specifier:
                members;
};
```

In this case, the Derived class inherits the members of the Base_class in private mode. So, all the members of the Base_class become private in Derived class. Thus,

Protected Members become Private Members
Public Members become Private Members

b. Protected Inheritance

General form

```
class Derived: protected Base_Class
{
  access specifier:
    members;
};
```

In this case, the Derived class inherits the members of the Base_class in protected mode. So, the protected and public members of the Base_class become protected in Derived class. Thus,

Protected Members become Protected Members
Public Members become Protected Members

c. Public Inheritance

General form

```
class Derived: public Base_Class
{
      access specifier:
              members;
};
```

In this case, the Derived class inherits the members of the Base_class in public mode. So, the public members of the Base_class become public in Derived class. Thus,

Protected Members become Protected Members
Public Members become Public Members

Note that the private members of a base class will never be accessible in the derived class.

C++ Inheritance and Class Scope

Every class in C++ has its own individual scope even when it inherits members from other classes. Those members that are explicitly defined in the class definition of a derived class belong to the scope of that class, those members which are members in virtue of inheritance belong to the scope of the super-class from which they were inherited. The scope rules for C++ classes have a number of consequences.

1) Those member functions which belong to the scope of a derived class will only have access to the data members within the scope of the derived and to those data members which are inherited from the public or protected section of the super-class.
2) Constructors for a derived type must make use of super-class constructors in order to initialise the data members inherited from the super-class.

Where the data members of a superclass should not be accessible for all but accessible to members of the subclass, those members should be declared in the protected section of the superclass.

4.2 Types of Inheritance

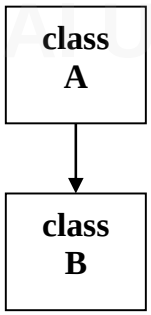
The inheritances seen so far come under the category of single inheritance. There are five different types of inheritance namely:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

In this type of inheritance a new class called the ***derived class*** inherits the members of a single class called the ***base class***. There is only one level of inheritance. This means that the newly created class does not have any derived class and has only a single parent class.

This is diagrammatically represented as:

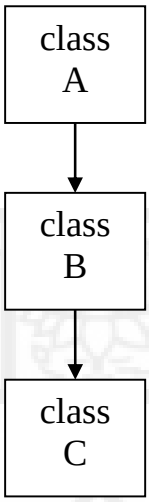


Single Inheritance

In this case class A is the super class of class B.

Multi-level inheritance

In this type of inheritance a new class called the derived class inherits the members of a single class called the base class. This newly created class becomes a parent class for another class. Thus, there exists more than one level of inheritance. This is diagrammatically represented as:

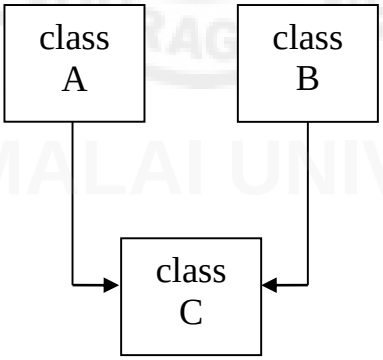


Multi-level inheritance

In this case class A is the super class of class B, which in turn is the base class of class C.

Multiple Inheritance

In this type of inheritance a new class called the derived class inherits the members of more than one class. This is diagrammatically represented as follows:

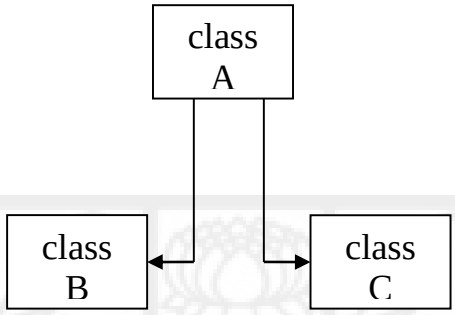


Multiple inheritance

In this case, class C inherits the members of both class A as well as class B. Thus class C has two parent classes.

Hierarchical Inheritance

In this type of inheritance more than one new class inherits the members of a single base class. This is diagrammatically represented as follows:

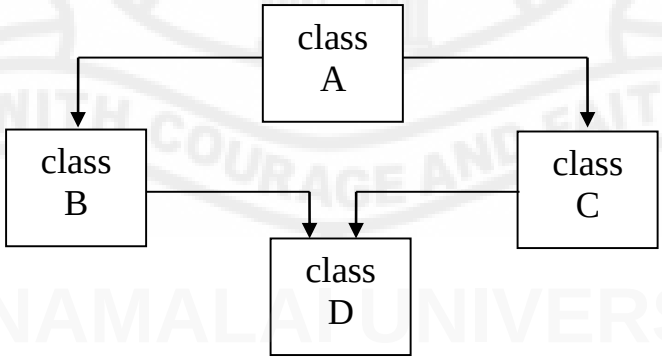


Hierarchical inheritance

In this case, two classes B and C inherit the members of class A.

Hybrid Inheritance

A combination of any two or more types of inheritance is called ***hybrid*** inheritance.



Hybrid Inheritance

In this case, class B and C inherit the members of class A through hierarchical inheritance and these two classes (B and C) are in turn inherited by D through multiple inheritance.

Multiple Inheritance

Single inheritance implies that a class has only one direct 'super-class'; Multiple inheritance allows the derivation of a class from more than one base class. Consider a class as follows:

```
#include<iostream.h>
        class Add
        {
            int x,y;
          public:
            Add()
         {
           cout<< "Constructor Inside Add class"<<endl;
           cout<<  "Enter two numbers";
           cin>>x>>y;
        }
        void sum()
        {
          cout<< "Sum"<<(x+y)<<endl;
        }
      };
        class Subtract
        {
           int a,b;
           public:
          Subtract()
         {
           cout<< " Constructor Inside Subtract class"<<endl;
           cout<< "Enter two numbers";
           cin>>a>>b;
         }
        void diff()
        {
          cout<< "Difference"<<(a-b)<<endl;
        }
      };

        class Math : public Add, public Subtract
        {
           public:
           Math()
          {
            cout<< " Constructor Inside Math class"<<endl;
           }
         };
         main()
```

```
        {
          Math m;
          m.sum();
          m.diff();
        }
```

Output:

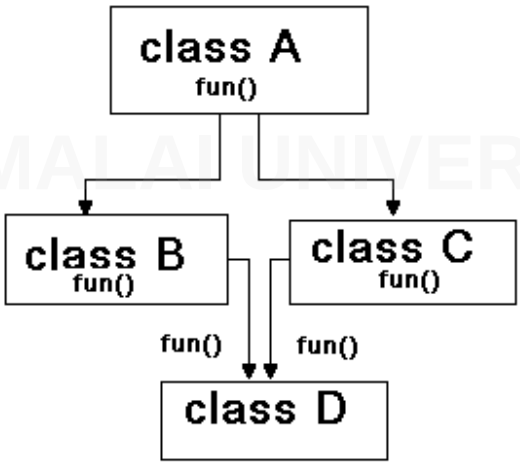
```
Constructor Inside Add class
Enter two numbers
3
4
 Constructor Inside Subtract class

Enter 2 numbers
5
6
 Constructor Inside Math class
Sum7
Difference-1
```

The class ***Math*** has two base classes ***Add*** and ***Subtract***. By creating an object of class ***Math*** the members ***sum()*** and ***diff()*** of class ***Add*** and ***Subtract*** respectively can be called. However, using an object of class ***Add***, the function ***diff()*** cannot be accessed and similarly using an object of class ***Subtract*** the function ***add()*** cannot be accessed.

4.3 Virtual Base Class

A potential problem for multiple inheritance occurs when a chain of inheritance derives from a single class via two or more paths, thus causing duplication of inherited data and functions.



As can be seen from the diagram, classes ***B*** and ***C*** inherit the member function fun() of class ***A*** using hierarchical inheritance. Class ***D*** inherits the members of class ***B*** and class ***C*** through multiple inheritance. Ultimately, class ***D*** will have 2 copies of the function fun()- one inherited from class ***B*** and the other inherited from class ***C***. While trying to access the function fun() in ***D***, the compiler will not know which function to access although both functions are the same. This will result in compilation error.

To avoid this problem the keyword ***virtual*** is included in the inheritance list:

```
class A
{
public:
   int memberA; //etc.
};
class B : virtual public A
{
// B members ...
};
class C : virtual public A
{
// C members ...
};
class D : public B, public C
{
// has access to one and only one memberA
}
```

To illustrate this consider the following example:

```
#include<iostream.h>
class Grand
{
        protected:
        int data;
public:
  Grand()
   {
    data =1;
   }

};

class Mother : public Grand
{
};
class Father  : public Grand
{
```

```
};
class Child : public Mother, public Father

{
public:
        int getdata()
{
  return data; // This is an error since the term data is ambiguous
}
};

void main()
{
Child c;
cout << c.getdata();
}
```

Output:

None- error in name resolving

In this case the class ***Child*** has two copies of the variable data of class ***Grand***. One it inherits through the class ***Father*** and the other through the class called class ***Mother***. When the function ***getdata()*** of the ***Child*** class is called the compiler gets confused as it would not know which of the data it has to call, although both are the same. This problem can be fixed by using the keyword ***virtual*** when deriving class ***Mother*** and class ***Father*** from class ***Grand***. The program is thus modified as follows:

Example

```
#include<iostream.h>
class Grand
{
        protected:
        int data;
 public:
     Grand()
       {
         data =1;
       }
};
class Mother : virtual public Grand
{};
class Father  : virtual public Grand
{};
class Child : public Mother, public Father
{
public:
        int getdata()
```

```
        {
            return data;
        }
};

    void main()
    {
        Child c;
        cout << c.getdata();
    }
```

Output:

```
1
```

The virtual keyword tells the compiler to inherit only one sub-object into subsequent derived classes.

4.4 Pointers

Pointers can be confusing and at times, one may wonder why one would ever want to use them. The truth is, they can make some things much easier. For example, using pointers is one way to have a function modify a variable passed to it; it is also possible to use pointers to dynamically allocate memory, which allows certain programming techniques, such as linked lists.

Pointers point to locations in memory. Picture a big jar that holds the location of another jar. In the other jar holds a piece of paper with the number 12 written on it. The jar with the 12 is an integer and the jar with the memory address of the 12 is a pointer

Pointer syntax can also be confusing, because pointers can both give the memory location and give the actual value stored in that same location. When a pointer is declared, the syntax is this: variable_type *name; Notice the *. This is the key to declaring a pointer and it is used before the variable name.

There are two ways to use the pointer to access information about the memory address it points to. It is possible to have it give the actual address to another variable. To do so, simply use the name of the pointer without the *. However, to access the actual value in that particular memory location pointed by the pointer, use the *. The technical name for doing this is dereferencing.

In order to have a pointer actually point to another variable it is necessary to have the memory address of that variable also. To get the memory address of the variable, put the & sign in front of the variable name. This helps in retrieving the address of that variable. This is called the reference operator, because it returns the memory address.

Example

```
#include <iostream.h>
int main()
{
            int x;
            int *pointer;
           pointer =&x;
           cout<<"Enter the input  :  ";
           cin>>x;
          cout<<*pointer;

          return 0;
}
```

Output:

```
Enter the input : 3
3
```

The **cout** outputs the value in x. Why is that? Well, look at the code. The integer is called x. A pointer to an integer is then defined as "pointer". Then it stores the memory location of x in pointer by using the ampersand (&) symbol. If you wish, you can think of it as if the jar that had the integer had an ampersand **(&)** in it then it would output its name (in pointers, the memory address) Then the user inputs the value for x. Then the **cout** uses the * to put the value stored in the memory location of pointer. If the jar with the name of the other jar in it had a * in front of it, then it would give the value stored in the jar, the name of which is stored in this jar. It is not too hard, the * gives the value in the location. The un astricked gives the memory location.

Notice that in the above example, pointer is initialized to point to a specific memory address before it is used. If this was not the case, it could be pointing to anything. This can lead to extremely unpleasant consequences to the computer. one should always initialize pointers before you use them.

The keyword **new** is used to initialize pointers with memory from free store (a section of memory available to all programs). The syntax looks like the example:

Example

```
int *ptr = new int;
```

It initializes **ptr** to point to a memory address of size **int** (because variables have different sizes, number of bytes, this is necessary). The memory that is pointed to becomes unavailable to other programs. This means that the careful coder will free this memory at the end of its usage.

The delete operator frees up the memory allocated through new. To do so, the syntax is as in the example.

Example

```
delete ptr;
```

After deleting a pointer, it is a good idea to reset it to point to **NULL**. **NULL** is a standard compiler-defined statement that sets the pointer to point to, literally, nothing. By doing this, you minimize the potential for doing something foolish with the pointer.

4.5 Pointers to Objects

As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

item x;

where item is a class and **x** is an object defined to be of type item. Similarly one can define a pointers it_ptr of type item as follows:

item *it_ptr;

Object pointers are useful is creating object at run time. Pointer is used to access the public members of an object. Consider a class item defined as follows:

```
class item
    {
        int code;
        float price;
    public:
        void getdata(int a, float b)
        {
            code = a;
            price = b;
        }
        void show(void)
        {
            cout << "code :" << code << "\n" << "Price:" << price << "\n\n";
        }
    };
```

Lets us declare an item variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of x.

One can refer to the member functions of item in two ways, one by using the dot operator and the object, and another by using the arrow operator and the object pointer. The statements,

```
x.getdata(100,75.50);
x.show();
```

are equivalent to

```
ptr→getdata(100,75.50);
ptr→show();
```

Since ***ptr** is an alias of **x**, one can also use the following method:

(*ptr).show();

The parenthesis is necessary because the dot operator has higher precedence than the indirection operator *.

Create the objects using pointers and **new** operator as follows:

item *ptr = **new** item;

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr**. Then **ptr** can be used to refer to the members as shown below:

ptr→show();

If a class has a constructor with arguments and does not include an empty constructor, one must supply the arguments when the object is created.

Create an array of objects using pointers. For Example, the statement

item *ptr = new item[10];//array of 10 objects

create a memory space for an array of 10 objects of **item**. Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

The following example illustrates the use of pointers to objects. One object is assigned to another object rather than passing the values as an argument.

```
#include <iostream.h>

class item
{
        int code;
        float price;
  public:
        void getdata(int a, float b)
  {
        code = a; price = b;
  }
  void show(void)
  {
        cout << "Code  : " << code << "\n";
         cout << "Price : " << price << "\n";
  }
};
const int size = 2;
main()
{
```

```
        item *p = new item [size];
        item *d = p;
        int x, i;
        float y;
        for(i=0;i<size;i++)
        {
                cout << "Input code and Price for Item  :  " << (i+1);
                cin >> x >> y;
                 p->getdata(x,y);
                 p++;
          }
           for(i=0; i<size;i++)
            {
                cout << "Item : " << (i+1) << "\n";
                    d->show();
                    d++;
              }
}
```

Output:

```
Input code and Price for Item 1 :  1001 5000
Input code and Price for Item  2 :  1002 10000
Item : 1
Code  : 1001
Price : 5000
Item : 2
Code  : 1002
Price : 10000
```

4.6 'this' Pointer

C++ uses unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object calls this function. For example, the function call **A.max()** will set the pointer **this** to the address of the object **A**. The starting address is the same as the address of the first variable in class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the member functions. Consider the following simple example:

```
class ABC
{
        int a;
        . . . .
        . . . .
};
```

The private Variable a can be used directly inside a member function, like

a = 123;

The following statement is used to do the same job;

this -> a = 123;

Since C++ permits the use of shorthand form a = 123, programmers are not using the pointer in this explicit manner. However, user have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, one passes only one argument to the function. The other argument is implicitly passed using the pointer **this**. One important application of the pointer **this** is to return the object it points to. For example, the statement

return *this;

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member functions and return the *invoking object* as a result. Example:

```
person  greater(person x)
{
        if x.age > age;
                return x;       // argument object
        else
                return *this    // invoking object
```

Invoke the above function by the call,

max = A.greater(B);

This function will return the object **B** (argument object) if the age of the person **B** is greater than of **A,** otherwise, it will return the object **A** (invoking object) using the pointer **this**. Remember, the dereference operator * produces the contents at the address contained in the pointer. A complete program to illustrate the use of **this** is given below.

```
#include <iostream.h>
#include <string.h>

class person
{
  char name[20];
  int age;

public:
  person (char *s, int a)
  {
        strcpy(name, s);
```

```
    age = a;
  }
  person greater(person x)
  {
        if(x.age >= age)
        return x;
   else
        return *this;
  }

  void display(void)
  {
        cout << "Name :" << name <<"\n";
        cout << "Age  :" << age << "\n";
   }
};

main()
{
        person P1("Arun",37), P2("Kumar",30), P3("Subash",40);
         person P(" ",0);
        P = P1.greater(P3);
        //P3.greater(P1)
       cout << "Elder person is:\n";
       P.display();
       P = P1.greater(P2);       //P2.greater(P1)
        cout << "Elder Person is :\n";
        P.display();
        getch();
}
```

Output:

```
Elder person is:
Name :Subash
Age  :40.549999
Elder Person is :
Name :Arun
Age  :37.5
```

4.7 Pointers to Base and Derived Classes

A pointer to a base class can be assigned the address of one of the base's derived class objects. If the derived class overrides members of the base, the compiler associates pointer reference accesses with the base class components of the object. This means that if a derived class member overrides a base class member, the pointer ignores the override.

```
#include<iostream.h>
class Base
{
    public:
         void fun_base()
     {
              cout << "\nBase";
      }
 };
class derv : public Base
{
    public:
         void fun_der()
         {
             cout << "\nDerv1";
         }
 };
  void main()
  {
     derv dv1;
     derv *ptr;
      ptr=&dv1;
       ptr->fun_base();

  }
```

Output:

Base

4.8 Introduction to Virtual Functions

The same function name is used in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, c++ determines which function to use at run time based on the type of objects pointed to by the base pointer, rather than the type of the pointer.

When a virtual function is called from a pointer, the compiler does not always know what type of object the pointer is pointing to. In this case, it may not know which version of the function to call at compile time and this judgment is delayed until runtime. At runtime, when the function call is executed, the code finds out the type of object whose address is in the pointer and calls the appropriate function, depending on the class of the object.

Static Binding

As in traditional programming languages, the compiler calls fixed functions based on the code. The linker replaces the identifiers with a physical address. Essentially, all function calls are resolved at compilation itself. Connecting to the functions in the normal way at compilation is called **early or static binding.**

Statbind.cpp

```
#include<iostream.h>
class Base
{

public:
        void show()
        {
          cout << "\nBase";
        }
};

void main()
{
        Base b;
        b.show();
}
```

Output:

```
Base
```

Late Binding

Selecting a function during runtime is called **late or dynamic binding.** In this program, the compiler does not know which version of show() to call since it depends on which type of object ptr points to. It could be an object of derv1 or derv2 class. At runtime the program sees which type of object ptr points to and accordingly calls either derv1::show() or derv2::show.

dynabind.cpp

```
#include <iostream.h>
class Base
{
    public:
        virtual void show()
        {
            cout << "\nBase";
        }
```

```
};
class derv1 : public Base
{
 public:
          void show()
          {
            cout << "\nDerv1";
          }
};
class derv2 : public Base
{
 public:
          void show()
          {
                cout << "\nDerv2";
          }
};
void main()
{
  derv1 dv1;
  derv2 dv2;
  Base* ptr;
  ptr = &dv1;
  ptr-> show();
  ptr = &dv2;
  ptr->show();
}
```

Output:

```
Derv1
Derv2
```

Late binding increases the power and flexibility of the program. It is also used to perform the same kind of action on different objects. However it requires some overhead and excessive use of late binding results in loss in speed and reduced runtime efficiency of code.

4.9 Abstract Classes

Abstract classes are classes from which no objects will be instantiated, but serve as the base for derived classes. They are enforced in C++ using pure virtual functions. A class that contains at least one pure virtual function is said to be an abstract class since it contains one or more functions for which there are no definitions. No object of an abstract class can be created. An abstract class is designed only to act as a base class, upon which other classes may be built.

```
#include<iostream.h >
class A
{
  public:
          virtual void print() = 0;
};

void main()
{
     A *a;
     a = new A; // This object cannot be created
     a->print();
}
```

Program will result in an error

Example

```
#include <iostream.h>

class number
{
  protected:
            int value;
  public:
            virtual void show() = 0; // pure virtual function

            void Setvalue(int v)
           {

            }
};
class hexa : public number
{
      public:
           void show()
           {
             cout << "The number is : " << value << endl;
           }
};

void main()
{
           hexa h;
           h.Setvalue(15);
           h.show();
}
```

Output:
The number is: 1

Since the class number has a function show() assigned zero (without code), it is called as an abstract class and it cannot be instantiated.

Pure Virtual Functions

A pure virtual function is a virtual function with no body. This is so when there is no need for the base class version of a particular function. The declaration of the pure virtual function has an assignment-to-zero notation to indicate that there is no definition for the function.
The prototype for this is

virtual return_type Function_name(parameter list)=0;

When a derived class does not redefine a virtual function, the version defined in the base class will be executed. In situations like this, a base class may not be able to define an object sufficiently to allow a base class virtual function to be created or like all derived classes override a virtual function. To handle these kind of cases, C++ supports pure virtual functions.

The compiler requires each derived class to either define the function or re declare it as a pure virtual function. A class containing pure virtual functions cannot be used to declare any objects of its own. If the derived class fails to override the pure virtual function then an error is encountered.

Example

```
#include <iostream.h>

class number
{
  protected:
            int value;
  public:
            virtual void show() = 0; // pure virtual function

             void Setvalue(int v)
            {
                        value = v;
              }
};

class hexa : public number
{
public:
          void show()
          {
```

```
                cout << "The number is : " << hex << value << endl;
        }
};

void main()
{
          hexa h;
          h.Setvalue(15);
           h.show();
}
```

Output:

The number is: f

Here show() is a pure virtual function and is not defined completely in the class number. It is defined only in the subclass hex. As a result no instance of the class number can be created (since it is not complete) and number is called an abstract class.

4.10 Summary

- Inheritance is the mechanism that allows a class **A** to inherit properties of another class **B**.
- The class from which a new class inherits is called the Super Class and the newly created class is called Derived Class.
- A new class can inherit the members of a base class in ***public***, ***protected*** or ***private*** modes.
- Single inheritance is the mechanism by which a class inherits the members of only one class.
- Multiple inheritance is the mechanism by which a class inherits the members of more than one class.
- Pointer is a variable that can hold the address of another variable.
- "this" Pointer contains the address of the current object through which the function is being invoked.
- Dynamic Binding/Late Binding is a process in which function calls are resolved at runtime.
- Static Binding/Early Binding is a process in which the identifiers are associated with the physical address during the process of compilation and linkage.
- A Virtual Function is a function, which does not exist but nevertheless appears real to some parts of the program.
- A Virtual Functions is declared as virtual in base class with the keyword **virtual** and redefined in the derived class.
- An Abstract Class is a class, which has at least one pure virtual function.
- An Abstract class cannot be instantiated.
- A Pure Virtual Function is a function declared in a base class that has no definition relative to the base class.
- The declaration of pure virtual function has an assignment-to-zero notation.

4.11 Review Questions

1. What does inheritance mean in C++?
2. Describe the syntax of multiple inheritance. When is the inheritance used?.
3. What is virtual Base class?
4. When is the class made virtual?
5. Inheritance helps in making a general class into a more specific class? (True/False).
6. A base class is never used to create objects? (True/ False).
7. Define class person (A person has a name, address, telno, email, DOB etc). Extend that class to define a classes called Official, Student, Staff each of them having specific properties. Implement virtual functions for getting details and calculating the performance.
8. Explain this pointer with an example.
9. Explain pure virtual function with an example.

Unit V

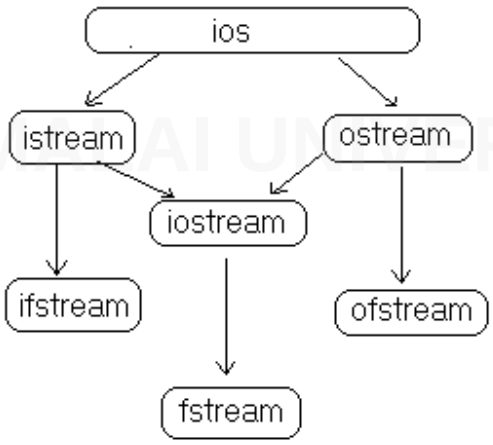
5.0 Objective

- Introduction to File class hierarchy
- Opening and closing of files
- Random Access File
- Command Line Argument

5.1 Introduction to File Class and Hierarchy

Files are used to store information permanently and to retrieve it whenever required. The file handling techniques of C++ support file manipulation in the form of stream objects. A ***stream*** is a general name given to the flow of data in an ***input/output*** process. In fact ***cin*** and ***cout*** used so far are themselves stream objects. These stream objects are pre-defined in the header file ***iostream.h***. Both ***cin*** and ***cout*** are used for standard input and output respectively. The ***extraction*** operator '>>' is a member of the ***istream*** class and the ***insertion*** operator '<<' is a member of the ***ostream*** class. Both of these classes are derived from ***ios*** class.

Disk files require a different set of classes than files used with standard input and output. C++ has two basic classes to handle files, ***ifstream*** and ***ofstream***. ***ifstream*** handles file input (reading from files) and ***ofstream*** handles file output (writing to files). There is another class called ***fstream*** which can handle both input and output. All these classes are present in the header file ***fstream.h***. Hence this header file has to be included for programs pertaining to disk file handling. The following diagram illustrates the stream class hierarchy.



Stream Class Hierarchy

5.2 Opening and Closing of files

open()

The first operation generally done on an object of one of the stream classes is to associate it to a real file, that is to say, to open a file. The function used for this purpose is ***open()***. The general syntax of open() is:

void open (const char * *filename***,** *openmode***);**

where *filename* is a string of characters representing the name of the file to be opened and *openmode* is a combination of the following flags:

`ios::in`	Open file for reading
`ios::out`	Open file for writing
`ios::ate`	Erase file before reading or writing
`ios::app`	Every output is appended at the end of file
`ios::binary`	Binary mode

The class ***ios*** is a ***virtual base class*** for the ***istream*** and ***ostream***. The ***iostream***(input and output stream) is a class with multiple inheritance from both ***istream*** and ***ostream***. These flags can be combined using bitwise operator OR (`|`). For example, if a file "example.bin" is to be opened in binary mode to add data it can be written as follows:

```
ofstream file;
file.open ("example.bin", ios::app | ios::binary);
```

Suppose no mode is specified, this function will directly open an already existing file in read mode when used with an object of ***ifstream*** and in write mode when used with an object of ***ofstream***. The member functions ***open*** of classes ***ofstream***, ***ifstream*** and ***fstream*** include a default mode when opening files that vary from one to other:

Class	Default mode to parameter
`ofstream`	`ios::out`
`ifstream`	`ios::in`
`fstream`	`ios::in | ios::out`

The default value is only applied if the function is called without specifying a mode parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined.

close()

When reading, writing or data manipulation operations on a file are complete the file has to be closed so that it can be used for some other application. The function

close()is used for this purpose and it terminates the connection between the stream and buffer. The general syntax is:

void close ();

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other process.

eof()

The function *eof()* returns true (non-zero value) if end-of-file is encountered while reading a file. Otherwise the function returns a false (zero) value. This function is generally used while reading from a file.

Writing a text to a file

The following program illustrates opening a file and writing a set of lines to the specified file.

```
//storing strings of text on a text file
#include <fstream.h>
void main()
{
        ofstream outfile;
        outfile.open ("outfile.txt");
        outfile << "This is my first file"<<endl;
        outfile.close();
}
```

Note:

A file called "outfile.txt" with the specified text will be created. No output will appear on the monitor.

The program first creates an object of ***ofstream***. The ***open()*** function is called using this object and a file by name ***outfile.txt*** is created. To write into this file the insertion operator is used along with the object of ***ofstream***. This is very similar to writing into standard output except for the fact that the output is now directed to the file instead of the monitor. The last statement

outfile.close();

disconnects the file document from the output stream outfile.The following program accepts the input from the user and writes into a file. The program terminates when the user inputs the character '0'.

```
#include <fstream.h>
void main()
{
        ofstream of;
        char c;
        of.open ("abc.txt");
        cin>>c;
        while(c!= '0')
        {
          of << c;
         cin>>c;
         }
        of.close();
}
```

The above program first accepts an input from the user. It checks if that character is not equal to zero. If it is not, it enters the while loop and writes that character into a file. This way, it accepts all the characters one by one and writes into the file until the user inputs a zero.

Reading a text from a file

The following program illustrates how to open a file, read its contents and display it on the screen. The program accepts a file name from the user. When the user supplies a file name, the program checks if such a file exists and the program exits if the file does not exist. The function ***fail()*** is used for this purpose and returns true if the file is not opened.

```
//Reading the contents of a text file and printing it out
#include <fstream.h>
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main()
{
        ifstream infile;
        char fname1[20];
        char ch;
        cout << "Enter the filename to be read \n";
        cin >> fname1;
        infile.open(fname1);
        if (infile.fail())
        {
                cerr << "No such file exists \n";
                exit(1);
        }
        while (!infile.eof())
```

```
 {
        infile>>ch;
        cout<<ch;
 }
        infile.close();

  } // end of program
```

Output:

```
Enter the filename to be read
outfile.txt
This is my first file
```

The program uses an object of ***ifstream*** for opening the file, as the contents from the file are to be read. The program uses a while loop to check if all the contents of the file are read. The expression "***!infile.eof()***" will return true as long as the end of file is not reached. If the '***!***' symbol is missed out the program will not read anything from the file. Thus the while loop ensures that all the characters till the end of the file are read. Inside the loop the object of ***ifstream*** is used along with the extraction operator to read the content of the file. This is again similar to ***cin*** except that the contents are now read from the file instead of keyboard. Once a character from the file is read it is printed on the monitor using the ***cout*** statement. Care should be taken to ensure that both these statements appear inside the loop as otherwise the output will not be proper.

It is always not necessary to use the ">>" operator for reading from a file. A function called ***getline()*** can also be used for this purpose. The following program uses a ***getline()*** function for reading from a file. The function ***getline()*** copies the content of a file into a character array which is then printed.

```
// reading a text file
#include <iostream.h>
#include <fstream.h>

int main () {
  char buffer[256];
  ifstream examplefile ("outfile.txt");

  while (! examplefile.eof() )
  {
    examplefile.getline (buffer,100);
    cout << buffer << endl;
  }
 }
```

Output:

```
This is my first file
```

Reading and Writing objects to a file

C++ supports features for writing to and reading from the disk file objects directly. The binary input and output functions ***read()*** and ***write()*** are designed to handle the entire structure of an object as a single object, using the computer's internal representation of data.

These functions are included in the File streams. The function ***write()*** is a member function of ***ostream***, also inherited by ***ofstream***. The function ***read()*** is member function of ***istream*** and it is inherited by ***ifstream***. Objects of class ***fstream*** have both the functions.

Their prototypes are:

*write (char * buffer, streamsize size);*
*read (char * buffer, streamsize size);*

where, buffer is the address of a memory block where read data are stored or where the data to be written are taken from.

The size parameter is an integer value that specifies the number of characters to be read to/written from the buffer.

5.2.1 write()

The ***write()*** member function is used to save the stream of objects on a specified file. The following program segment depicts how to write objects belonging to a class into a file using the ***write()*** function.

```
#include <fstream.h>
class emp_info
{
        private:
        char name[30];
        int age;
        public:
        void input_data()
        {
          cout<< "Enter the name :  "<<endl;
          cin>>name;
          cout<<  "Enter the age :  "<<endl;
          cin>>age;
        }
        void show_data()
        {
          cout<< "Name :"<<name<<endl;
          cout<< "Age    :"<<age;
        }
        }; //end of class definition
```

```
void main()
{
        emp_info z; //object z of the base class emp_info
        fstream f;
        z.input_data();
        f.open ("emp.txt",ios::out);
        f.write ((char*) &z, sizeof(z));
        f.close();
} // end of program
```

Output:

Enter the name:
Ashish
Enter the age:
20

Note:

This information will be written into a file called emp.doc

The program reads the already stored information from the file ***emp.doc***. The information from the file is first read from the file and then the ***show_data()*** function is called to print the name and age of the employee. Care should be taken that information has to be printed on the monitor only after accepting the value for the instance variables from the file. Note that the function ***input_data()*** is never called in this case.

read()

The ***read()*** member function is used to get data for a stream of objects from a specified file. The following program segment depicts how to read objects belonging to a class using the ***read()*** function.

```
//reading an object from a file
#include <fstream.h>
class emp_info
{
        private:
        char name[30];
        int age;
        public:
        void input_data()
        {
          cout<< "Enter the name"<<endl;
          cin>>name;
          cout<<  "Enter the age"<<endl;
          cin>>age;
        }
        void show_data()
```

```
        {
          cout<< "Name :"<<name<<endl;
          cout<< "Age  :"<<age;
        }
}; //end of class definition

void main()
{
  emp_info z; //object z of the base class emp_info
  fstream f;
  f.open ("emp.txt",ios::in);
  f.read ((char*) &z, sizeof(z));
  z.show_data();
  f.close();
} // end of program
```

Output:

```
Name: Ashish
Age    : 20
```

The program writes the object of class emp_info into a file called emp.doc. The class emp_info has two functions input_data() and show_data(). The input_data() function is first called to accept the name and age of the employee. Once this information is got they are written down into the file emp.doc. Care should be taken that information has to be written into the file only after accepting the value for the instance variables.

5.3 Random Access File

The knowledge of the logical location at which the current ***read*** or ***write*** operation occurs is of great importance in achieving faster access to information stored in a file. Thus a file in which any record can be accessed at random is called a ***Random Access file***.

C++ I/O system supports four functions for setting a file pointer to any desired position inside the file or gets the current file pointer. These functions allow the programmer to have control over the position at which the ***read*** or ***write*** operations take place. The ***seekp()*** and ***tellp()*** are member functions of class ***ofstream***. The ***seekg()*** and ***tellg()*** are member functions of class ***ifstream***. All these four are available in ***fstream***. These functions can be used in manipulating the position of the stream pointers that point to the reading or writing locations within a stream.

tellg() and tellp()

These two member functions accept no parameters and return an integer value representing the current position of ***get*** stream pointer (in case of tellg) or ***put*** stream pointer (in case of tellp).

seekg() and seekp()

This pair of functions serves respectively to change the position of stream pointer's ***get*** and ***put***. The syntax is as follows:

seekg(offset, origin)
seekp(offset,origin)

Using this prototype, an offset from a concrete point determined by parameter direction can be specified. These functions move the associated file's pointer to offset number of bytes from the specified origin, which must be one of the following three values:

ios::beg	beginning of file
ios::cur	current location
ios::end	end of the file

The values of both stream pointers ***get*** and ***put*** are counted in different ways for text files than for binary files, since in text mode files, some modifications to the appearance of some special characters can occur.

The following program uses the functions ***seekg()*** and ***tellg()*** to count the number of bytes.

```
#include <iostream.h>
#include <fstream.h>
main () {
  long m;
  ifstream file ("outfile.txt", ios::in);
  file.seekg (0, ios::end);
  m = file.tellg();
  file.close();
  cout << "size of " << "outfile.txt";
  cout << " is " << (m) << " bytes.\n";
  }
```

Output:
size of outfile.txt is 10 bytes.

Imagine the file outfile.txt contains the following content
Hi 12345

Initially the file is opened in read mode. The ***tellg()*** function returns the current position of the stream pointer which will obviously be the first position and hence will return 0.

The statement

file.seekg (0, ios::end);

will move the pointer to the extreme end of the file and the ***tellg()*** function at that stage will return the current position of the pointer. This value will be a count of the actual number of characters stored in the file.

The following program demonstrates the use of ***seekp()*** function in writing a set of characters at a specified location of the file.

```
#include <fstream.h>
void main()
{
        char num[10];
        fstream inout("new.txt", ios::out| ios::in);
        for (int i=0;i<10;i++)
                inout << i;
        inout.seekp(4);
        char var[5];
        cout << "Enter a string of 4 characters: ";
        cin >> var;
        inout <<var;
        inout.seekg(2);
        inout>>num;
        cout << num <<endl;
}
```

Output:

```
Enter a string of 4 characters: good
23good89
```

The program creates a file called ***new.txt*** and writes into it all the numbers from 0 to 9 using a for loop. The statement,

inout.seekp(4);

positions the pointer at the 4^{th} location (numbering of location starts from 0). The program then accepts a four letter string which is then written starting from that particular location. Hence the number 4567 gets replaced by good (the entered string). The file ***new.txt*** then contains the following content

0123good89

The program then starts reading the content of the file from location 2 which produces the output as:

23good89

Advantages of C++ stream classes

- The stream classes form a powerful set of classes. They can be modified, extended or expanded to incorporate user-defined data types or classes.
- They offer a rich set of error handling facilities.
- They are fully buffered, and thereby they reduce disk access.
- They encapsulate their internal working from the user. The programmer need not specify the type of data to be input or output. It is determined automatically by the stream class.

5.4 Command line arguments

In C++ it is possible to accept command line arguments. To do so, one must first understand the full definition of int main (). It actually accepts two arguments, one is the number of command line arguments and the other is a listing of the command line arguments.

It looks like this:

int main(int argc, char* argv[])

The integer argc, is the Argument Count (hence argc). It is the number of arguments passed into the program from the command line, including the path and name of the program.

The array of character pointers is the listing of all the arguments. **argv[0]** is entire path to the program including its name. After that, every element numbers less than **argc** are command line arguments. One can use each **argv** element just like a string or use **argv** as a two dimensional array.

```
#include<iostream.h>
void main(int argc,char* argv[])
 {
  cout<<" Argc: " <<argc;
  for(int i=0;i<argc;i++)
  cout<<" Argv: " <<argv[i];
  }
```

***Output* :(when run in dos prompt)**

```
   writefile 4 pop 45 67
 Argc: 5 Argv: C:\BC5\BIN\WRITEFILE.EXE Argv: 4 Argv: pop Argv: 45 Argv: 67
```

Where **writefile** is the name of the file.

To read the command-line arguments the main() function must itself be given two arguments. The first argc represents the total number of command line arguments. The first command-line argument is always the pathname of the current program. The remaining command-line arguments are those typed by the user.

5.5 Summary

- A stream is a sequence of characters and is connected to a device or a file.
- To send/receive data to/from an external file the file must be connected to a stream.
- C++ supports both input and output with files through the following classes:
  - ofstream: File class for writing operations (derived from ostream)
  - ifstream: File class for reading operations (derived from istream)
  - fstream: File class for both reading and writing operations (derived from iostream)
- Object can be written into a file using the functions ***read()*** and ***write()***.
- A random access file helps in accessing a particular record.

5.6 Review Questions

1. Write a note on stream class hierarchy.
2. Write a note on reading and writing text to a file.
3. Write a program that creates a file called ***Info.txt*** to write a string into it.
4. Write a program that reads the content of a file that has the following content.
   "*Reading the information from the file*"
5. Write a program that writes an object of a class into a file and reads the same.
6. Explain command line argument with an example.