ANNAI WOMEN'S COLLEGE

(Arts & Science)

(Affiliated to Bharathidasan University, Tiruchirappalli - 620 024)

TNPL Road, Punnam Chatram, Karur - 639 136.

Faculty Name : Dr. T. Pavithra

Major : BCA **Semester : II**
Subject Code : 16SCCCA2 **Subject : Programming in C++**

SYLLABUS

PROGRAMMING IN C++

UNIT-I

Principles of object oriented programming – Beginning with C++ - Tokens, Expressions and control structures – Functions in C++.

UNIT – II

Classes and objects – Constructors and destructors – New operators – Operator overloading – Type conversion.

UNIT - III

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

UNIT - IV

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

UNIT - V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

UNIT-I

Principles of object oriented programming – Beginning with C++ - Tokens, Expressions and control structures – Functions in C++.

Principles of object oriented programming

BASIC CONCEPTS OF OOP:

Object-oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

The important concepts of OOPs are:

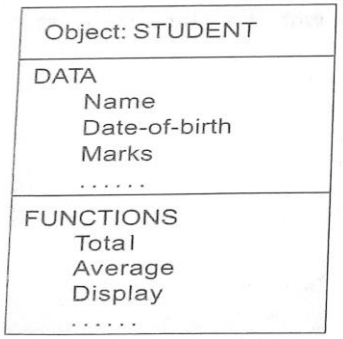
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

❖ **Objects:**

→ Objects are the basic run-time entities in an object-oriented system. Each object that contains data, and functions (code to manipulate the data).

→ They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

→ When a program is executed, the objects interact by sending messages to one another.



Classes:

→ A ***class*** is a collection of objects of similar type. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

→ Objects are like variables of the class.

→ Once a class has been defined, we can create any number of objects belonging to that class.

→ For example, mango, apple and orange are objects of the class **fruit**.

❖ **Data Abstraction and Encapsulation:**

→ **Abstraction:** *Abstraction* refers to the act of representing essential features without including the background details or explanations.

→ The classes use the concept of data abstraction known as Abstract Data Types (ADT).

→ **Encapsulation:** The wrapping up of data and functions into a single unit (called class) is known is known as *encapsulation.*

→ The insulation of the data from direct access by the program is called ***data hiding*** or ***information hiding***.

→ These functions provide the interface between the object's data and the program.

❖ **Inheritance:**

→ *Inheritance* is the process by which objects of one class acquire the properties of objects of another class.

→ It supports the concept of hierarchical classification.

→ **For example,** the bird is a part of the class 'flying bird' which is again a part of the class 'bird'.

→ The principle behind this is that each derived class shares common characteristics with the class from which it is derived.

→ The concept of inheritance provides the idea of *reusability.*

→ We can add additional features to an existing class without modifying it.

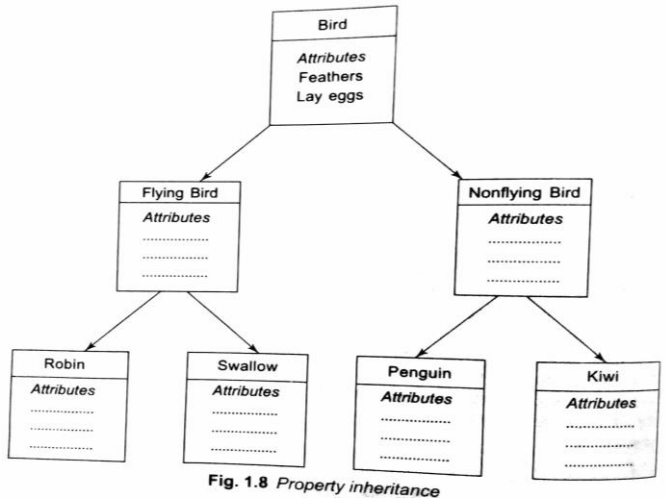


Fig. 1.8 *Property inheritance*

❖ **Polymorphism:**

→ *Polymorphism* means the ability to take more than one form.

→ Poly (many), morph (forms)

→ Using a single operator to perform different types of operations is known as *operator overloading.*

→ Using a single function name to perform different types of tasks is known as *function overloading.*

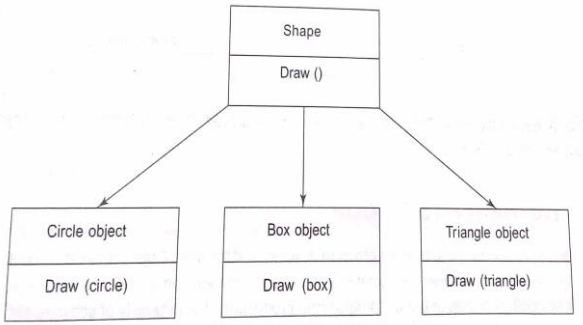


Fig. 1.9 *Polymorphism*

❖ **Dynamic Binding:**

→ *Binding* refers to the linking of a procedure call to the code to be executed in response to the call.

→ *Dynamic binding* (late binding) means that the code associated with a given procedure call in not known until the time of the call at run-time.

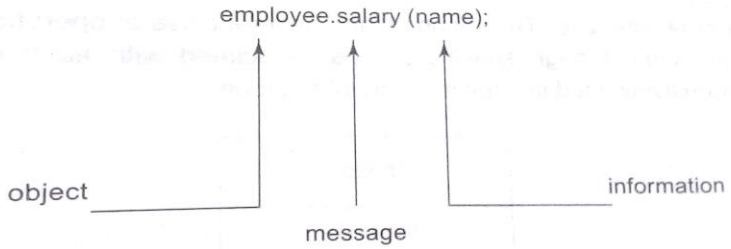
→ At run-time, the code matching the object under current reference will be called.

❖ **Message Passing:**

→ An object-oriented programming consists of a set of objects that communicate each other.

→ The objects can communicate with one another by message passing.

→ Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.



OOP offers several benefits to both the program designer and the user.

- The redundancy can be eliminated by using inheritance.
- The development time can be reduced and the productivity can be increased by using standard working modules.
- The data hiding helps the programmer to build secure programs.
- It is possible to create multiple instances of an object without any interference.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much easier.
- Software complexity can be easily managed.

OBJECT-ORIENTED LANGUAGES:

A language that is specially designed to support the OOP concepts makes it easier to implement them.

There are two categories of languages that support OOP:

1. Object-based programming languages
2. Object-oriented programming languages

- *Object-based* programming is the style of programming primarily supports encapsulation and object identity.

  Major features:

  ✓ Data encapsulation
  ✓ Data hiding and access mechanisms
  ✓ Automatic initialization and clear-up of objects
  ✓ Operator overloading
  ✓ Ex: Ada

- *Object-oriented* programming incorporates all of object-based programming along with inheritance and dynamic binding.

  OOP=object-based features + inheritance + dynamic binding

  Ex: C++, Smalltalk, Java

APPLICATIONS OF OOP:

✓ Real-time systems
✓ Simulation and modeling
✓ Object-oriented databases
✓ AI and expert systems
✓ Hypertext, hypermedia and expertext

- ✓ Neural networks and parallel programming
- ✓ Decision support and office automation systems
- ✓ CIM/CAM/CAD systems

BEGINNING WITH C++

Introduction:

C++ is an object oriented programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs.

C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX

Initially named the new language in "c with classes" later in 1983 the name was changed into C++ (Idea of increment version of C)

STRUCTURE OF C++ PROGRAM

The structure of C++ program is divided into four different sections:

1. **Header File Section**
2. **Class Declaration section**
3. **Member Function definition section**
4. **Main function section**

(1) Header File Section:

- ✓ This section contains various header files.
- ✓ You can include various header files in to your program using this section.
- ✓ For example: # include <iostream.h >
- ✓ Header file contains declaration and definition of various built in functions as well as object.

(2) Class Declaration Section:

- This section contains declaration of class.
- You can declare class and then declare data members and member functions inside that class.

SYNTAX:

```
class classname
{
Member declarations;
Function declarations;
};
```

For example:

```
class Demo
{
int a, b;
public:
void input();
void output();
};
```

(3) Member Function Definition Section:

- ✓ This section is optional in the structure of C++ program.
- ✓ Because you can define member functions inside the class or outside the class.
- ✓ If all the member functions are defined inside the class then there is no need of this section.
- ✓ This section is used only when you want to define member function outside the class.
- ✓ This section contains definition of the member functions that are declared inside the class.

For example:

```
void Demo:: input ()
{
cout << "Enter Value of A:";
cin >> a;
cout << "Enter Value of B:";
cin >> b;
}
```

(4) Main Function Section:

n this section you can create an object of the class and then using this object you can call various functions defined inside the class as per your requirement.

For example:

```
Void main ()
{
Demo d1;
d1.input ();
d1.output ();
}
```

TOKENS

The smallest individual units in a program are called tokens. The tokens are:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language.

❖ KEYWORDS:

Keywords are reserved words and they have predefined meaning. They can't be used as user-defined program elements. Etc…

Ex: class, while, switch, for,if,else

❖ IDENTIFIERS AND CONSTANTS:

- **Identifier** refers to the name of a variable, a function, an array, or a class.

Naming rules:

- Only alphabetic characters, digits and underscores are permitted.
- The first letter must be an alphabet or an underscore.
- Uppercase and lowercase letters are distinct.
- Keywords cannot be used as an identifier.
- There is no limit on the length.
- White spaces are not allowed.
- Ex:
  - name, first_name, name_1, _123 are **valid** identifiers.
  - amount$, 12a, first name are **invalid** identifiers.

- **Constants** refer to fixed values that do not change during the execution of a program.
  - Constants do not have memory locations.
  - Ex: 123, 12.34, "C++", '\0'

❖ STRING:

A string is a sequence of characters. In C++, character array is used to store a string.

Ex: **"Welcome"**

❖ OPERATORS:

An operator is a symbol which is used to perform the given mathematical operations. There are many types of operators in C++. These can be broadly categorized as: arithmetic, relational, logical, bitwise, assignment and other operators.

Arithmetic Operators

Assume variable A holds 10 and variable B holds 20, then −

- + Adds two operands. A + B will give 30
- - Subtracts second operand from the first. A - B will give -10
- * Multiplies both operands. A * B will give 200
- / Divides numerator by de-numerator. B / A will give 2

- **%** Modulus Operator and remainder of after an integer division. B % A will give 0
- ++ Increment operator, increases integer value by one. A++ will give 11
- -- Decrement operator, decreases integer value by one. A-- will give 9

Relational Operators

Assume variable A holds 10 and variable B holds 20, then −

- == Checks if the values of two operands are equal or not, if yes then condition becomes true. (A == B) is not true.
- **!=** Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. (A != B) is true.
- > Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. (A > B) is not true.
- < Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. (A < B) is true.
- >= Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. (A >= B) is not true.
- <= Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. (A <= B) is true.

Logical Operators

Assume variable A holds 1 and variable B holds 0, then −

- **&&** Called Logical AND operator. If both the operands are non-zero, then condition becomes true.(A && B) is false.
- || Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.(A || B) is true.
- **!** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows.

- **&** Binary AND Operator copies a bit to the result if it exists in both operands.(A & B) will give 12 which is 0000 1100
- | Binary OR Operator copies a bit if it exists in either operand.(A | B) will give 61 which is 0011 1101
- ^ Binary XOR Operator copies the bit if it is set in one operand but not both.(A ^ B) will give 49 which is 0011 0001
- ~ Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
- << Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.A << 2 will give 240 which is 1111 0000
- >> Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.A >> 2 will give 15 which is 0000 1111

Assignment Operators

- = Simple assignment operator, Assigns values from right side operands to left side operand.C = A + B will assign value of A + B into C
- += Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.C += A is equivalent to C = C + A
- -= Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.C -= A is equivalent to C = C - A

- ***=** Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.C *= A is equivalent to C = C * A
- **/=** Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.C /= A is equivalent to C = C / A
- **%=** Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.C %= A is equivalent to C = C % A
- **<<=** Left shift AND assignment operator.C <<= 2 is same as C = C << 2
- **>>=** Right shift AND assignment operator.C >>= 2 is same as C = C >> 2
- **&=** Bitwise AND assignment operator.C &= 2 is same as C = C & 2
- **^=** Bitwise exclusive OR and assignment operator.C ^= 2 is same as C = C ^ 2
- **|=** Bitwise inclusive OR and assignment operator.C |= 2 is same as C = C | 2

EXPRESSIONS AND THEIR TYPES

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values.

An expression may consist of one or more operands, and zero or more operators to produce a value.

Types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

Constant Expressions: Constant Expressions consist of only constant va1ues.

Examples:

15
20 + 5 / 2.O
'x'

Integral Expression: Integral Expressions are those which produce results after implementing all the automatic and explicit type conversions.

Examples:

m
m * n - S
m * ' x'
5 + int(2.0) where m and n are integer variables.

Float Expressions: Float Expressions are those which, after all conversions, produce floating-point results.

Examples:

x + y
x * y / 10
5 * float (10)
10.75

where x and y are floating-point variables.

Pointer Expressions: Pointer Expressions produce address values.

Examples:

&m
Ptr

ptr + l where m is a variable and ptr is a pointer.

Relational Expressions: Relational Expressions yield results of type bool which takes a value true or false. Relational expressions are also known as Boolean expressions.

Examples:

x<= y

a+b == c+d

m+n > 100

Logical Expressions: Logical Expressions combine two or more relationa1 expressions and produces bool type results.

Examples:

a>b && x== 10

x ==10 || y== 5

Bitwise Expressions: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Shift operators are often used for multiplication and division by powers of two.

Examples:

x << 3 // Shift three bit position to left

y >>1 // Shift one bit position to right

Special Assignment Expressions:

Chained Assignment: A chained statement can not be used to initialize variables at the time of declaration.

x=(y=10); or x=y=10;

First 10 is assigned to y and then to x.

Embedded Assignment

x = (y = 50) + 10 ;

Compound Assignment

C++ supports a compound assignment operator. The general form of the compound assignment operator is:

variable1 op= variable2;

Example:

x = x + 10; can be written as **x += 10;**

The operator += is known as compound assignment operator or short-hand assignment operator.

CONTROL STRUCTURES

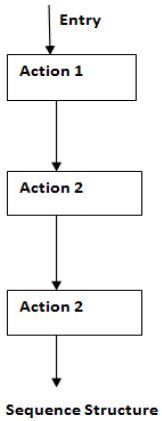
Introduction

Control structures form the basic entities of a "structured programming language". Control structures are used to alter the flow of execution of the program.

There are three types of control structures available in C and C++

1. Sequence structure (straight line paths)
2. Selection structure (one or many branches)
3. Loop structure (repetition of a set of activities)

Sequence structure:



Sequence Structure

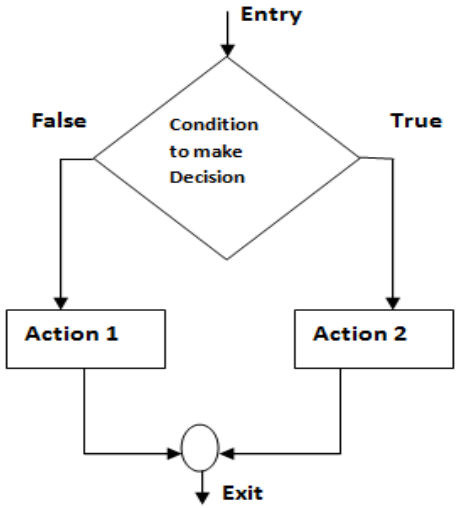
Selection Structure:

Selection structures are used to perform 'decision making' and then branch the program flow based on the outcome of decision making.

Selection structures are implemented in C/C++ with If, If Else and Switch statements.

If and If Else statements are 2 way branching statements where as Switch is a multi branching statement.

i) Simple if
ii) if else
iii) else if ladder
iv) Nested if
v) switch statement



Selection Structure

Implemented using:- If and If..else control statements

switch is used for multi branching

Simple if:

This expression is evaluated. If expression is TRUE statements inside the braces will be executed

```
if (expression)
{
statement 1;
statement 2;
}
```

if else:

Expression 1 is evaluated. If TRUE, statements inside the curly braces are executed. If FALSE program control is transferred to immedate else if statement.

Syntax:

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

Else if:

Expression 1 is evaluated. If TRUE, statements inside the curly braces are executed.

If FALSE program control is transferred to immediate else if statement.

If expression 1 is FALSE, expression 2 is evaluated. If expression 2 is FALSE, expression 3 is evaluated.

If all expressions (1, 2 and 3) are FALSE, the statements that follow this else (inside curly braces) is executed.

Syntax:

```
if(expression 1)
 {
statement 1;
 }
else if(expression 2)
{
statement 2;
}
else if(expression 3)
{
statement 3;
}
else
{
statement 4;
}
```

Switch statement: Switch is a multi branching control statement. Case is the keyword used to match the integer/character constant from expression. value1, value2 ... are different possible values that can come in expression. break is a keyword used to break the program control from switch block.

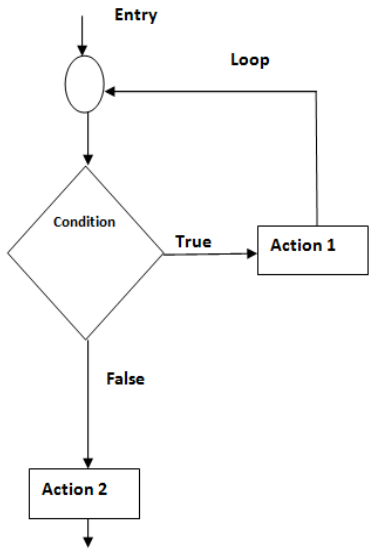
Syntax:

```
switch(expression)
{
case value1:
statement 1;
break;
case value2:
statement 2;
break; …….
default:
statement n;
break;
}
```

Loop Structure:

A loop structure is used to execute a certain set of actions for a predefined number of times or until a particular condition is satisfied.

There are 3 control statements available in C/C++ to implement loop structures. **While, Do while and For statements.**



Loop Structure

Implemented using:- While , Do While and For control statements

While loop:

This condition is tested for TRUE or FALSE. Statements inside curly braces are executed as long as condition is TRUE.

```
while(condition)
{
statement 1;
statement 2;
}
```

do while loop:

```
do
{
statement 1;
statement 2;
}
while(condition);
```

The for statement

```
for(initialization statements; test condition; iteration statements)
{
statement 1;
statement 2;
}
```

FUNCTIONS IN C++

Introduction

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main().

We can divide up program into separate functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Function Declaration: A function declaration tells the compiler about a function name and how to call the function.

The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list )
{
body of the function
}
```

A C++ function definition consists of a **function header** and a **function body**. Here are all the parts of a function −

Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, and number of the parameters of a function.

Function Body − The function body contains a collection of statements that define what the function does.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

EXAMPLE:

```
#include <iostream.h>
int max(int num1, int num2);
int main ()
{
  int a = 100;
  int b = 200;
  int ret;
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
  return 0;
}
int max(int num1, int num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

OUTPUT:

Max value is : 200

Methods of calling function

1. **Call by Value:** This method copies the actual value of an argument into the formal parameter of the function.
2. **Call by Pointer:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call.
3. **Call by Reference:** This method copies the reference of an argument into the formal parameter. Reference is used to access the actual argument used in the call.

INLINE FUNCTIONS:

- An Inline function is a function this is expanded in line when it is

invoked.

- The compiler replaces the function call with the corresponding function when it is invoked.
- It is used to eliminate the cost of calls to small functions. Because, every time a function is called, It takes a lot of extra time in executing the tasks such as jumping to the function, saving return address into the stack, and returning to the calling function.
- **Syntax**

```
inline function-header
{
function body
}
```

- To make a function inline, the inline keyword is added in the function header.
- In some situations, inline functions may not work. They are:
  - ✓ For a function returning values. If a loop, a switch, or a goto exists.
  - ✓ For functions not returning values, if a return statement exists.
  - ✓ If functions contain static variables.
  - ✓ If inline functions are recursive.

//Inline Function Example

```
#include<iostream.h>
inline int max(int a,int b)
{
if(a>b)
  return(a);
else
  return(b);
}
void main()
{
int a=10,b=20;
cout<<"\n Big="<<max(a,b);
}
```

Output:

Big=20

DEFAULT ARGUMENTS:

- C++ allows us to call a function without specifying all its arguments.
- The function assigns a default value to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared.
- Advantages:
  - We can use default arguments to add new parameters to the existing functions.
  - Default arguments can be used to combine similar functions into one.

//Default Arguments Example

```
#include<iostream.h>
void printline(char c='*',int count=15)
{
cout<<"\n";
for(int i=1;i<=count;i++)
  cout<<c;
}
void main()
```

Output:

```
***************
 Hai.
################
 You are welcome to C++ Lab.
```

```
{
printline();
cout<<"\n Hai.";
printline('#');
cout<<"\n You are welcome to C++ Lab.";
printline('$',30);
}
```

FUNCTION OVERLOADING:

- Overloading refers to the use of the same thing for different purposes.
- The same function name can be used to perform a variety of tasks is referred to as Function overloading. It is also known as function polymorphism.

- Ex:

Function prototype	Function call
int add(int a, int b);	cout<<add(5,10);
int add(int a, int b, int c);	cout<<add(5,10,15);
double add(double x, double y);	cout<<add(12.5, 7.5);

//Function overloading example

```
#include<iostream.h>
int add(int a)
{
  return(a+a);
}
int add(int a,int b)
{
  return(a+b);
}
float add(float a,float b)
{
  return(a+b);
}
void main()
{
cout<<"\nFunction Overloading";
cout<<"\nOne int arg\n"<<add(10);
cout<<"\nTwo int args\n"<<add(10,20);
cout<<"\nTwo float args\n"<<add(11.22f,33.44f);
}
```

Output:

```
Function Overloading
One int arg
20
Two int args
30
Two float args
44.66
```


UNIT – II

Classes and objects – Constructors and destructors – New operators – Operator overloading – Type conversions

CLASSES AND OBJECTS

- A class is a way to bind the data and its associated functions together.
- It hides the data from external use, if necessary.
- It is used to create a user-defined data type. (Abstract data type)
- A class specification has two parts:
  - Class declaration – Describes the type and scope of its members.
  - Class function definition – Describes how the class

functions are implemented.

The general form of a class declaration is:

```
class class_name
{
 private:
        variable declarations;
        function declarations
 public:
         variable declarations;
        function declarations;
};
```

- The keyword 'class' specifies, that what follows is an abstract data of type class_name.
- The body of a class is enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members.* The variables are called as **data members** and the functions are called as **member functions**.
- They are grouped under two sections, namely, ***private*** and ***public*** to denote which of the members are private and which of them are public.
- The keywords private and public are known as **visibility labels.**
- The private members can be accessed only from within the class. It is called **data hiding.**
- The public members can be accessed from outside the class also.

Creating objects and accessing class members:

- ✓ Once a class has been defined, we can create variables of type class by using class name.
- ✓ The class variables are known as **objects.**
- ✓ The declaration of object is similar to that of a variable of any basic data type.
- ✓ Ex:

```
class person
{
    char name[20];
    char place[20];
public:
    void read();
    void display();
};
```

- ✓ To access the class members, the **dot operator** is used.
  - Syntax:

```
object_name.member_name;
```

Ex:

```
x.read();
y.display();
```

//Classes and Objects Example

```
#include<iostream.h>
class person
{
 char name[20];
 char place[20];
public:
```

```
 void read()
 {
  cout<<"\n Enter name and place\n";
  cin>>name>>place;
 }
 void display()
 {
  cout<<"\n Name: "<<name;
  cout<<"\n Place: "<<place;
 }
};
void main()
{
cout<<"\n Classes and objects \n";
person p;        // Creating objects
p.read();
p.display();
}
```

Output:

```
Classes and objects

Enter name and place
Rose
Pattukkottai

Name: Rose
Place: Pattukkotai
```

FRIEND FUNCTION:

- A nonmember function cannot have an access to the private data members of a class.
- The friend function is used to access the private data of a class where it is declared.
- To make a function friendly to a class, the function declaration should be preceded by the keyword **friend**.
- Special Characteristics of a friend function:
  - It is not in the scope of the class to which it has been declared as friend.
  - It cannot be called using the object of that class.
  - It can be invoked like a normal function without the help of any object.
  - It cannot access the member names directly, and has to use an object name and dot membership operator with each member name.
  - It can be declared either in the public or the private part of a class.
  - Usually, it has the objects as arguments.

//Friend function example

```
#include<iostream.h>
class B;
class A
{
int a;
public:
A(int x)
{
a=x;
}
friend void add(A,B);
};
class B
{
int b;
public:
B(int x)
```

Output

```
Friend function

Sum=30
```

```
{
b=x;
}
friend void add(A,B);
};
void add(A obj1,B obj2)
{
cout<<"\n Sum="<<obj1.a+obj2.b;
}
void main()
{
cout<<"\n Friend function\n";
A obj1(10);
B obj2(20);
add(obj1,obj2);
}
```

CONSTRUCTORS

A constructor is a special member function whose task is to initialize the objects of its class.

The function name is same as the class name.

A constructor is invoked whenever an object of its associated class is created.

Characteristics of the constructor function:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, even void. They cannot return values.
- They cannot be inherited.
- Like normal functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- They make 'implicit calls' to the operators 'new' and 'delete' when memory allocation is required.

Types:

- **Default Constructor:** A constructor that accepts no parameters (arguments) is called the **default constructor.** If there is no such constructor, the compiler supplies a default constructor.
  - **Ex: add a;**
- **Parameterized Constructors:** Like a normal function, C++ permits us to pass arguments to a constructor. These constructors are called as parameterized constructor.
  - **Ex: add a(10,20);**
- **Copy constructor:** A copy constructor is used to declare and initialize an object from another object. The process of initializing through a copy constructor is known as *copy initialization.* A copy constructor takes to an object of the same class as itself as an argument.
  - **Ex:**

```
add a(10,20);
add a1;
a1=a;       //Copy constructor
```

****

```
//Constructors example
#include<iostream.h>
class add
```

```
{
  int a,b;
public:
  add()   //Default constructor
  {
    a=0;
    b=0;
  }
  add(int m,int n)  //Argumented Constructor
  {
    a=m;
    b=n;
  }
  add(add obj)    //Copy constructor
  {
    a=obj.a;
    b=obj.b;
  }
  void sum()
  {
    cout<<"\n Sum="<<a+b;
  }
};
void main()
{
add a1;
add a2(10,20);
add a3;
a3=a2;
a1.sum();
a2.sum();
a3.sum();
}
```

Output:

Constructors

Sum=0

Sum=30

DESTRUCTORS

- A destructor is used to destroy the objects that have been created by a constructor.
- The function name is same as the class name and preceded by a tilde (~).
- A destructor never takes any argument. It does not return any value. It will be invoked implicitly by the compiler upon exit from the program.
- It is used to clean up storage that is no longer accessible.
- The delete operator is used to free the allocated memory.
- The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed.

//Destructors example

```
#include<iostream.h>
class add
{
  int a,b;
public:
  add(int m,int n)
  {
    a=m;
    b=n;
  }
  void sum()
  {
```

Output:

Destructors

Sum=30

Sum=50

Deleting objects

Deleting objects

```
    cout<<"\n Sum="<<a+b;
  }
  ~add()
  {
    cout<<"\n Deleting objects";
  }
};
void main()
{
cout<<"\n Destructors \n";
add a1(10,20);
add a2(20,30);
a1.sum();
a2.sum();
}
```

NEW OPERATORS IN C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators.

1. **I/O operators**
   a. **<< → Insertion operator (input)**
   b. **>> → Extraction operator (output)**
2. **:: → Scope resolution operator**
3. **Member dereference operators**
   a. **::*** →**Pointer-to-member declarator**
   b. **->*** → **Pointer-to-member operator**
   c. **.*** →**Pointer-to-member operator**
4. **Memory management operators**
   a. **Delete → Memory release operator**
   b. **new → Memory allocation operator**
5. **Manipulators**
   a. **Endl → Line feed operator**
   b. **setw → Field width operator**

Scope Resolution Operator:

Scope resolution operator **(::)** in C++ is used to define a function outside a class or when we want to use a global variable but also has a local variable with the same name.

Syntax:

: : variable_name;

Example:

```
#include<iostream.h>
int m=10;
int main()
{
int m=20;
{
int k=m;
int m=30;
cout<<" we are in inner block \n";
```

OUTPUT:

```
We are in inner block
k=20
m=30
: :m=10
We are in outer block
m=20
: :m=10
```

```
cout<<"k="<<k<<"\n";
cout<<"m="<<m<<"\n";
cout<<"::m="<<::m<<"\n";
}
cout<<"we are in outer block \n";
cout<<"m="<<m<<"\n";
cout<<"::m="<<::m<<"\n";
return 0;
}
```

Manipulators:

Manipulators are the operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw.**

endl operator:

The **endl** operator used in an output statement, causes a linefeed to be inserted. It has the same effect using the newline character "\n". ex: **cout<<"welcome"<<endl;**

setw operator:

The **setw** operator is used to set a common field width for the data. **Setw(field width). Ex: cout<<setw(5)<<sum;** here, the sum value is 345 then the value will be display in following format.

		3	**4**	**5**

Example:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
int basic=12000,
int allowance=150,
int total;
salary=basic+allowance;
cout<<"BASIC="<< setw(10)<<basic<<endl
    <<"ALLOWANCE="<<setw(10)<<allowance<<endl
    <<"TOTAL="<<setw(10)<<total<<endl;
return 0;
}
```

OUTPUT:

```
BASIC=*****12000
ALLOWANCE=*******150
TOTAL=*****12150
```

OPERATOR OVERLOADING

The mechanism of giving special meaning to an operator is known as operator overloading.

To define an additional task to an operator, a special function called operator function is used to define the function.

The general form of an operator function is:

```
return-type classname::operator op(argument list)
{
        // Function body
}
```

Where, return-type specifies the return type of the function. op – the operator to be overloaded. Operator function must be either member function or friend function.

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function **operator op**() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required

operations.

Rules for operator overloading:

1. Only existing operators can be overloaded. New operators cannot be overloaded.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded:
   - Membership operator (.)
   - Pointer-to-member operator (.*)
   - Scope resolution operator (::)
   - sizeof operator
   - Conditional Operator (?:)
5. We cannot use friend functions to overload certain operators.
   - = Assignment operator
   - ( ) Function call operator
   - [ ] Subscripting operator
   - -> Class member access operator
6. For unary operators, by means of a member function, there is no explicit argument. In case of a friend function, there is one argument.
7. For binary operators, by means of a member function, there is one explicit argument. In case of a friend function, there are two arguments.
8. When using binary operators through member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

//Operator overloading example

```
#include<iostream.h>
class complex
{
  float real,imag;
public:
   complex(){ }
   complex(float r, float i)
   {
    real=r;
    imag=i;
   }
   void display()
   {
     cout<<real<<"+i"<<imag;
   }
   complex operator +(complex);
};
complex complex::operator +(complex c)
{
complex c1;
c1.real=real+c.real;
c1.imag=imag+c.imag;
return(c1);
}
void main()
{
cout<<"\n Binary Operator Overloading\n";
```

Output

```
Binary Operator Overloading
C1=12+i56
C2=10+i20
C3=22+i76
```

```
complex c1(12.34,56.78);
complex c2(10.123,20.232);
complex c3;
c3=c1+c2;
cout<<"\n C1=";
c1.display();
cout<<"\n C2=";
c2.display();
cout<<"\n C3=";
c3.display();
}
```

TYPE CONVERSIONS:

Type cast is a basically conversion between from one data type to another data type. There are two types of conversion.

1. **Implicit conversion (Automatic conversion):** It done by the compiler on its own, without any external trigger form the user.

Example:

```
#include<iostream.h>
int main()
{
int m;
float x=3.141;
cout<<"implicit conversion"<<endl;
m=x;
cout<<"the value of m="<<m;
return 0;
}
```

Here, automatically convert **x (float)** to an **integer** before its value assigned to **m**.

2. **Explicit conversion:** This process is also called type casting and it is user-defined. The compiler does not support automatic type conversion for such data types. It can be done by TWO ways.

   a. **Conversion by assignment: type(expression)**

Ex:

```
int main()
{
double x=1.24;
int sum=(int)x+1;
cout<<"SUM="<<sum;
return 0;
}
```

OUTPUT: **SUM=2**

   b. **Conversion using casting operator:** C++ supports four types of casting.
      - Static cast
      - Dynamic cast
      - Const cast
      - Reinterpret cast

Situations:

Three types of situations might arise in the data conversion between incompatible data types.

- Conversion from **basic type to class type.**
- Conversion from **class type to basic type.**

- Conversion from **one class type to another class type.**

Conditions:

The casting operator function should satisfy the following conditions.

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

RECURSION

- A function calls itself is referred to as recursion.
- The function is called itself either directly or indirectly.
- The recursive function requires a conditional check, to avoid the recursive calls of function goes into an infinite loop.

// Recursion Example

```
#include<iostream.h>
#include<conio.h>
int fact(int n)
{  if((n==1)||(n==0))
  return(1);
else
  return(n*fact(n-1));   }
void main()
{    int n=5;
cout<<"\n Factorial of "<<n<<" is "<<fact(n);
 }
```

OUTPUT

```
Factorial of 5 is 120
```

TYPE CONVERSIONS

There are three types of data conversion between uncompatible types.

- Conversion from basic type to class type
- Conversion from class type to basic type
- Conversion from one class type to another class type.

Conversion from basic type to class type:

- In this type of conversion the source type is basic type and the destination type is class type.
- Basic type is converted into class type.
- The constructor can be used to perform type conversion during the object creation.

Conversion from class type to basic type:

- In this type of conversion the source type is class type and the destination type is basic type.
- Class type is converted into basic type.
- C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.
- The general form of an overloaded casting operator function,

```
operator typename()
{
//  Function statements
}
```

- usually referred to as a conversion function, is:

This function converts a class type data to *typename*.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Conversion from one class type to another class type:

This type of conversion between objects of different classes can be carried out by either a constructor or a conversion function.

Example:

```
/* Program to convert basic type to class type using constructor */
#include "iostream.h"
#include "conio.h"
```

OUTPUT

```
Enter time duration in minutes Basic
Type to ==> Class Type Conversion...
100
1: Hours(s)
40 Minutes
```

```
class Time
{
        int hrs,min;
        public:
         Time(int);
                void display();
};
Time :: Time(int t)
{
 cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
        hrs=t/60;
        min=t%60;
}

void Time::display()
{
        cout<<hrs<< ": Hours(s)" <<endl;
        cout<<min<< " Minutes" <<endl;
}
void main()
{
        clrscr();
                int duration;
        cout<<"Enter time duration in minutes";
    cin>>duration;

        Time t1=duration;
        t1.display();
        getch();
}
```

UNIT-III

INHERITANCE: EXTENDING CLASSES (Types of Inheritance)

- ✓ The mechanism of deriving a new class from an old one is called *inheritance (or derivation).*
- ✓ The old class is referred to as the *base class* and the new one is called the *derived class or subclass.*
- ✓ Inheritance promotes the reusability of data and functions.

Defining Derived Classes:

A derived class can he defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
..... //
·····// members of derived class
.. .. //
 };
```

- ✓ The colon indicates that the derived-class-name is derived from the base-class-name.
- ✓ The visibility-mode is optional and, if present, may be either private or public. The default visibility-mode is private.
- ✓ Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:

```
class ABC: private XYZ   //Private derivation
{
members of ABC
};

class ABC: public XYZ //Public derivation
{
members of ABC
```

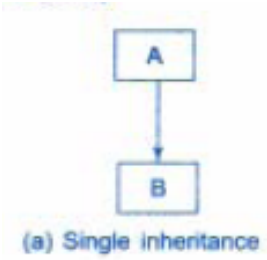
```
};
class ABC: XYZ     //Private derivation by default
{
members of ABC

};
```

- ✓ When a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- ✓ A public member of a class can be accessed by its own objects using the dot operator.
- ✓ When the base class is publicly inherited, 'public members' of the base class become •public members' of the derived class and therefore they are accessible to the objects of the derived class.
- ✓ In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.
- ✓ **Types:**
  - ➢ Single inheritance
  - ➢ Multilevel inheritance
  - ➢ Multiple inheritance
  - ➢ Hierarchical inheritance
  - ➢ Hybrid inheritance

Single Inheritance:

- ✓ A derived class with only one base class is called single inheritance.



(a) Single inheritance

Example:

```
/*****************************************
#include<iostream.h>
#include<conio.h>
class A
{
  int a;
public:
  void get_a()
  {
    cout<<"\n Enter a number ";
    cin>>a;
  }
  int put_a()
  {
    cout<<"\n a="<<a;
    return(a);
  }
};
class B: public A
{
  int b;
public:
  void get_b()
  {
    cout<<"\n Enter a number ";
```

OUTPUT

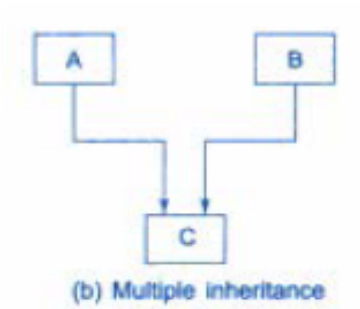
```
Enter a number
2
Enter a number
4

b=4
a=2
```

```
    cin>>b;
  }
  void put_b()
  {
    cout<<"\n b="<<b;
    cout<<"\n a+b="<<b+put_a();
  }
};
void main()
{
B bobj;
bobj.get_a();
bobj.get_b();
bobj.put_b();
}
*****************************************/
```

Multiple Inheritance:

- ✓ A class is derived from several base classes is called multiple inheritance.



(b) Multiple inheritance

The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...
{
      .....
      .....(Body of D)
      .....
};
```

where, vi1ibilily may be either public, private or protected. The base classes are separated by commas.

Example:

```
/*****************************************

#include<iostream.h>
#include<conio.h>
class A
{
  int a;
public:
  void get_a()
  {
    cout<<"\n Enter a number ";
    cin>>a;
  }
  void put_a()
  {
    cout<<"\n a="<<a;
  }
};
class B: public A
{
  int b;
public:
  void get_b()
  {
```

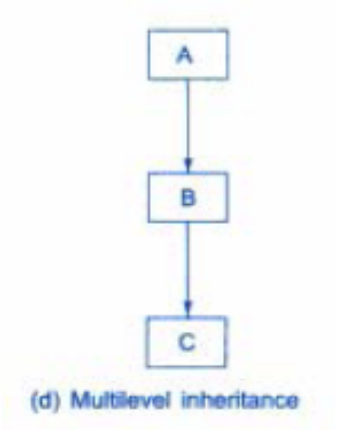
OUTPUT

```
Enter a number
1
Enter a number
2
Enter a number
3
a=1
b=2
c=3
```

```
    cout<<"\n Enter a number ";
    cin>>b;
  }
  void put_b()
  {
    cout<<"\n b="<<b;
  }
};
class C: public B
{
  int c;
public:
  void get_c()
  {
    cout<<"\n Enter a number ";
    cin>>c;
  }
  void put_c()
  {
    cout<<"\n c="<<c;
  }
};
void main()
{
C cobj;
cobj.get_a();
cobj.get_b();
cobj.get_c();
cobj.put_a();
cobj.put_b();
cobj.put_c();
}
*****************************************/
```

Multilevel Inheritance:

- ✓ The mechanism of deriving a class from another 'derived class is known as multilevel inheritance.



(d) Multilevel inheritance

- ✓ The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.
- ✓ The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as *inheritance path.*

Example:

```
/*****************************************

#include<iostream.h>
#include<conio.h>
class A
{
  int a;
public:
  void get_a()
```

```
  {
   cout<<"\n Enter a number ";
   cin>>a;
  }
  void put_a()
  {
   cout<<"\n a="<<a;
  }
};

class B: public A
{
  int b;
public:
  void get_b()
  {
   cout<<"\n Enter a number ";
   cin>>b;
  }
  void put_b()
  {
   cout<<"\n b="<<b;
  }
};
class C: public B
{
  int c;
public:
  void get_c()
  {
   cout<<"\n Enter a number ";
   cin>>c;
  }
  void put_c()
  {
```

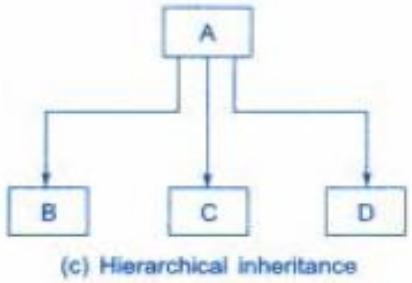
OUTPUT

```
Enter a number
1
Enter a number
2
Enter a number
3
a=1
b=2
c=3
```

```
   cout<<"\n c="<<c;
  }
};
void main()
{
C cobj;
cobj.get_a();
cobj.get_b();
cobj.get_c();
cobj.put_a();
cobj.put_b();
cobj.put_c();
}
************************************/
```

Hierarchical Inheritance:

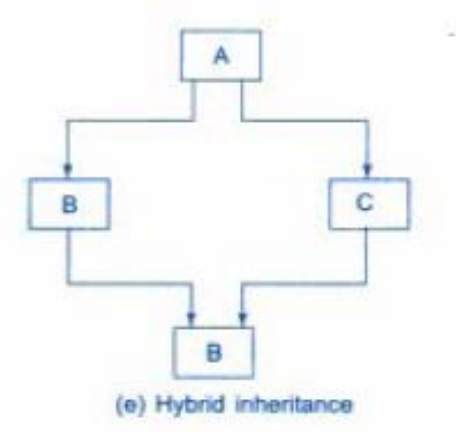
- ✓ More than one classes derived from one class. This process is known as *hierarchical inheritance.*



(c) Hierarchical inheritance

Hybrid Inheritance:

- ✓ The inheritance in which the derivation of a class involves more than one form of any inheritance is called **hybrid inheritance**.
- ✓ Basically **C++ hybrid inheritance** is combination of two or more types of inheritance. It can also be called multi path inheritance.



(e) Hybrid inheritance

****

TOKEN:

The smallest individual units in a program are called tokens. The tokens are:

6. Keywords
7. Identifiers
8. Constants
9. Strings
10. Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language.

❖ **KEYWORDS:**

Keywords are reserved words and they have predefined meaning. They can't be used as user-defined program elements.

Ex: class, while, switch

❖ **IDENTIFIERS AND CONSTANTS:**

- Identifier refers to the name of a variable, a function, an array, or a class.
- Naming rules:
  - Only alphabetic characters, digits and underscores are permitted.
  - The first letter must be an alphabet or an underscore.
  - Uppercase and lowercase letters are distinct.
  - Keywords cannot be used as an identifier.
  - There is no limit on the length.
  - White spaces are not allowed.
- Ex:
  - name, first_name, name_1, _123 are valid identifiers.
  - amount$, 12a, first name are invalid identifiers.
- Constants refer to fixed values that do not change during the execution of a program.
  - Constants do not have memory locations.
  - Ex: 123, 12.34, "C++", '\0'

❖ **STRING:**

- A string is a sequence of characters.
- In C++, character array is used to store a string.
- Ex: **"Welcome"**

❖ **OPERATOR:**

An operator is a symbol which is used to perform the given mathematical operations.

Ex: +, -, *, <, !

OPERATORS IN C++

- C++ has a rich set of operators.
- All C operators are valid in C++ also.
- In addition, C++ introduces some new operators. They are:
  - << - Insertion operator (input)
  - >> - Extraction operator (output)
  - :: - Scope resolution operator
  - ::* - Pointer-to-member declarator

- ->* - Pointer-to-member operator
- .* - Pointer-to-member operator
- delete - Memory release operator
- endl - Line feed operator
- new - Memory allocation operator
- setw - Field width operator

➢ **Scope Resolution Operator:**

- C++ is a block-structured language.
- The same variable name can be used to have different meanings in different blocks.
- The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.
- A variable declared inside a block is said to be local to that block.
- If same variable name is used in both the inner block and the outer block, the inner block hides the variable in the outer block.
- To resolve this problem C++ allows us to use the scope resolution operator (::).
- A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs.

```
/**********************************
// Scope resolution operator - Example
#include<iostream.h>
#include<conio.h>
int a=10;
void main()
{
  int a=100;
  cout<<"\n a="<<a;  //Access local variable a
  cout<<"\n ::a="<< ::a;  //Access global variable a
}
```

OUPUT

```
a=100

::a=10
```

**********************************/

Member Dereferencing Operators:

C++ permits us to access the class members through pointers. C++ provides a set of three pointer-to-member operators.

- ::* - To declare a pointer to a member of a class.
- ->* - To access a member using a pointer to the object and a pointer to that member.
- .* - To access a member using object name and a pointer to that member

Memory Management Operators

The new operator can be used to create objects of any type. It takes the following general form:

pointer-variable=new data-type;

Here pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to bold a data object of type data-type and returns the address of the object.

Example:

```
p = new int ;
q = new fl oat ;
```

When a data object is no longer needed. it is destroyed to release the memory space for reuse. The delete operator is used to delete the allocated space. The general form is:

delete pointer-variable;

Example:

```
delete p;
delete q;
```

MANIPULATORS:

- Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

- The endl manipulator, when used in an output statement causes a Linefeed to be inserted.
- It has the same effect as using the newline character "\n". For example, the statement

  cout << "m = " << m << end l;
- The setw manipulator is used to set a common field width for the data.

  Example:

  cout<< setw(S)<<sum<<endl;

TYPE CAST OPERATOR:

C++ permits explicit type conversion or variables or expressions using the type cast operator.

(type-name) expression // C notation
type-name (expression) // C++ notation

Examples:
average = sum/(float)i; // C notation
average = sum/float(i); // C++notation

****

EXPRESSIONS AND THEIR TYPES

- An expression is a combination of operators, constants and variables arranged as per the rules of the language.
- It may also include function calls which return values.
- An expression may consist of one or more operands, and zero or more operators to produce a value.

Types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

Constant Expressions:

Constant Expressions consist of only constant va1ues. Examples:

15
20 + 5 / 2.O
'x'

Integral Expressions:

Integral Expressions are those which produce results after implementing all the automatic and explicit type conversions.

Examples:

m
m * n - S
m * ' x'
5 + int(2.0)

where m and n are integer variables.

Float Expressions:

Float Expressions are those which, after all conversions, produce floating-point results.

Examples:

x + y
x * y / 10
5 * float(10)
10.75

where x and y are floating-point variables.

Pointer Expressions:

Pointer Expressions produce address values. Examples:

&m
ptr
ptr + l

where m is a variable and ptr is a pointer.

Relational Expressions:

Relational Expressions yield results of type bool which takes a value true or false. Examples:

x<= y
a+b = = c+d
m+n > 100

Relational expressions are also known as Boolean expressions.

Logical Expressions:

Logical Expressions combine two or more relationa1 expressions and produces bool type results.
Examples:

a>b && x = = 10
x = = 10 || y = = 5

Bitwise Expressions:

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.

Examples:

x << 3 // Shift three bit position to left
y >>1 // Shift one bit position to right

Shift operators are often used for multiplication and division by powers of two.

Special Assignment Expressions:

Chained Assignment

x={y=10) ;
or
x=y=10;

First 10 is assigned toy and then to x.

A chained statement can not be used to initialize variables at the time of declaration. For instance, the statement

float a = b = 12.34; // Wrong

Embedded Assignment

x = (y = 50) + 10 ;

Compound Assignment

C++ supports a compound assignment operator.

Example:

x = x + 10;

may be written as

x += 10;

The operator += is known as compound assignment operator or short-hand assignment operator.
The general form of the compound assignment operator is:

variable1 op= variable2;

****

VIRTUAL BASE CLASSES:

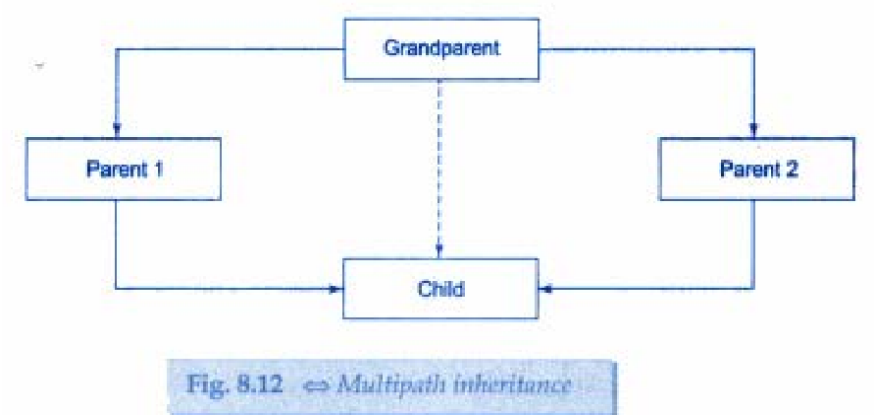


Fig. 8.12 ⇔ *Multipath inheritance*

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved.

This is illustrated in Fig. 8.12.

The 'child' has two direct base classes 'parent1' and 'parent2' which themselves have a common base class 'grandparent'.

The 'child' inherits the traits of 'grandparent.' via two separate paths. It can also inherit directly as shown by the broken line.

The 'grandparent' is sometimes referred to as indirect base class.

All the public and protected members of 'grandparentt' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'.

This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

Example:

```
class A          //grandparent
{
.....
.....
};
class B1: virtual public A     //parent 1
{
.....
.....
};
class B2 : public virtual A    //parent 2
{
.....
.....
};
class C : public B1, public B2 //child
{
.....
.....
};
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that classis inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

The keywords virtual and public may be used in either order.

Example:

```
/* ***************************
#include<iostream.h>
#include<conio.h>
class student {
   int rno;
public:
   void getnumber() {
      cout << "Enter Roll No:";
      cin>>rno;
   }
```

```
   void putnumber() {
      cout << "\n\n\tRoll No:" << rno << "\n";
   }
};

class test : virtual public student {
public:
   int part1, part2;

   void getmarks() {
      cout << "Enter Marks\n";
      cout << "Part1:";
      cin>>part1;
      cout << "Part2:";
      cin>>part2;
   }

   void putmarks() {
      cout << "\tMarks Obtained\n";
      cout << "\n\tPart1:" << part1;
      cout << "\n\tPart2:" << part2;
   }
};

class sports : public virtual student {
public:
   int score;

   void getscore() {
      cout << "Enter Sports Score:";
      cin>>score;
   }

   void putscore() {
      cout << "\n\tSports Score is:" << score;
   }
};

class result : public test, public sports {
   int total;
public:

   void display() {
      total = part1 + part2 + score;
      putnumber();
      putmarks();
      putscore();
      cout << "\n\tTotal Score:" << total;
   }
};

void main() {
   result obj;
   clrscr();
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
   getch();
}
```

OUTPUT

```
Enter Roll No: 200

Enter Marks

Part1: 90
Part2: 80
Enter Sports Score: 80

Roll No: 200
Marks Obtained
Part1: 90
Part2: 80
Sports Score is: 80
Total Score is: 250
```

ABSTRACT CLASSES

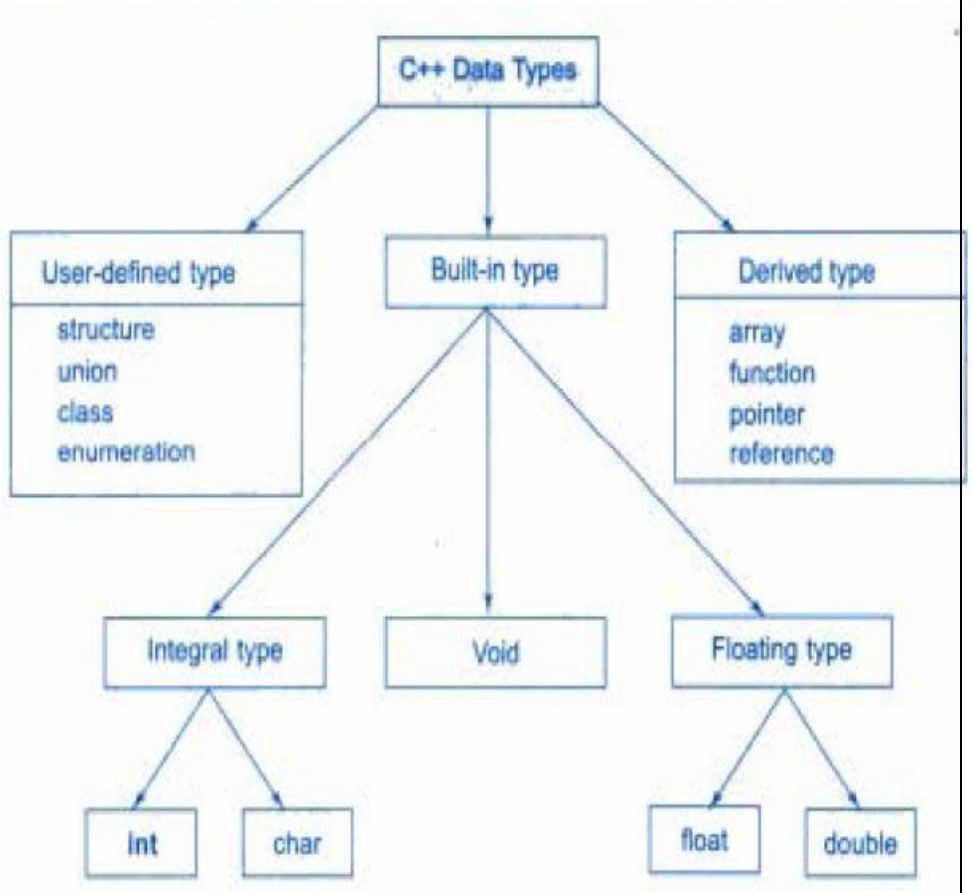
An abstract class is one that is not used to create objects.

An abstract class is designed only to act as a base class (to be inherited by other classes).

****

BASIC DATA TYPES:

Data types in C++ can be classified under various categories as shown below:



- C++ compilers support all the built-in (also known as basic or fundamental)
  data types.

- The basic data types may have several modifiers
  .
- The modifiers are signed, unsigned,
  long, and short may be applied to character and integer basic data types.

Basic Data Types:

- char - character data type. Used to store a single character. Size - 1 byte.
- int - integer data type. Used to store whole numbers. Size - 2 bytes.
- float - floating-point data type. Used to store real numbers. Size - 4 bytes.
- double - double data type. Used to store double precision floating-point number. Size - 8 bytes.
- void - The void data type is used to
  - to specify the return type of of a function when it is not returning any value.
  - to indicate empty argument list to a function.
    - Example: void func(void);
  - Used to declare a generic pointers. A generic pointer can be assigned a pointer value of any basic data type.
    - Example:

```
int *ip;
void *gp;
gp=ip;
```

User-Defined Data Types:

- **Structures and Classes**
  : structure is a collection of related data item.

  - Ex:

```
struct person
{
```

```
    char name[20];
    char place[20];
};
```

- **Class** is a collection of objects. A class is a way to bind the data and its associated functions together. It is similar to structure. But it contains the methods that operate on those data members.

Ex:

```
class person
{
   private:
        char name[20];
        char place[20];
  public:
        void getinfo();
        void dispinfo()
};
```

- **Type definition:** typedef is used to define another user defined data type. Then the new data type is used to declare variables of the new type.
  - Ex: typedef int mark;
  - mark tamil, english, major, allied;
- **Enumerated Data Type:**

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility or the code.

- Tho enum keyword automatically enumerates a list of words by assigning them values 0,1,.2. and so on.
- This facility provides an alternative means for creating symbolic constants.
- Examples:

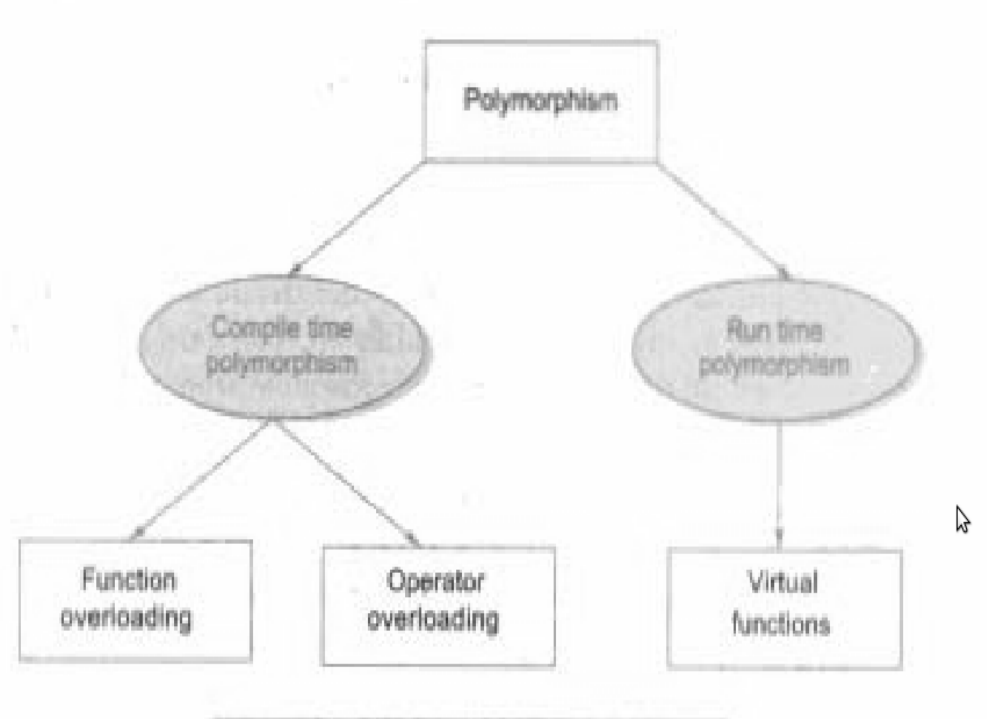
```
enum shape {circle. square, triangle) ;

enum colour (red, blue. green. yellow);
```

By using these names, new variables can be declared:

shape ellipse;

colour background;



Derived Data Types:

- **Arrays:** An array is a collection of data items of same type that share a common name.
  - Example: int a[20];
- **Functions:** Function is a group of statements that together perform a task.
  - Example: int fact(int);

- **Pointers:** Pointer is a variable that contains the address of another variable.
  - Example: int a,*p;
    p=&a;
- **Reference:** A reference variable provides an alias for a predefined variable.
  - Example:

int total;
int & sum=total; // sum is a reference variable.

****

POLYMORPHISM:

- Polymorphism means 'one name, multiple forms'. Same function is used for many purposes.
- In function overloading an object is bound to its function call at compile time. This is called compile time polymorphism. It is also known as early binding or static binding or static linking.
- In function overriding, the linking of function call to its definition is done at run time. This is called dynamic binding. It is also known as late binding. The virtual function is used to achieve the run time polymorphism.

POINTERS:

- Pointer is a variable that contains the address of another variable.
- **Declaring and initializing pointers:**

data-type *ptr-variable;

- A variable must be initialized before using it in a program.

Example:

```
int a,*p;
p=&a;
```

- The & (address of operator) is used to retrieve the address of a variable.

- **Pointer Expressions and Pointer Arithmetic**
  C++ allows pointers to perform the following arithmetic operations:
  - A pointer can be incremented(++) (or) decremented (- -)
  - Any integer can be added to or subtracted from a pointer
  - Tne pointer can be subtracted from another

  Example:

```
int a[6];
int *aptr;
aptr=&a[0];
aptr++
```

this Pointer:

- this pointer is used to represent the current object that invokes a member function**.**
- this is a pointer that points to the object for which this function was called.
- 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.

Example:

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

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

```
a = 123;
```

It can be also written as:

```
this -> a=123;
```

VIRTUAL FUNCTIONS:

- A virtual function a member function which is declared within base class and is re-defined (overridden) by derived class.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference used for function call.
- They are mainly used to achieve runtime polymorphism.
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at run-time.
- **Rules for virtual functions:**
  - The virtual functions must be members of some class.
  - They cannot be static members.
  - They accessed by using object pointers.
  - A virtual function can be a friend of another class.
  - A virtual function in a base class must be defined, even though it may not be used.
  - The prototypes of the base class version of a virtual function and all the derived class versions must be identical.
  - We cannot have virtual constructors, but we can have virtual destructors.
  - If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure Virtual Functions:

The function inside the base class is seldom used for performing any task. It only serves as a placeholder. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display() = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class.

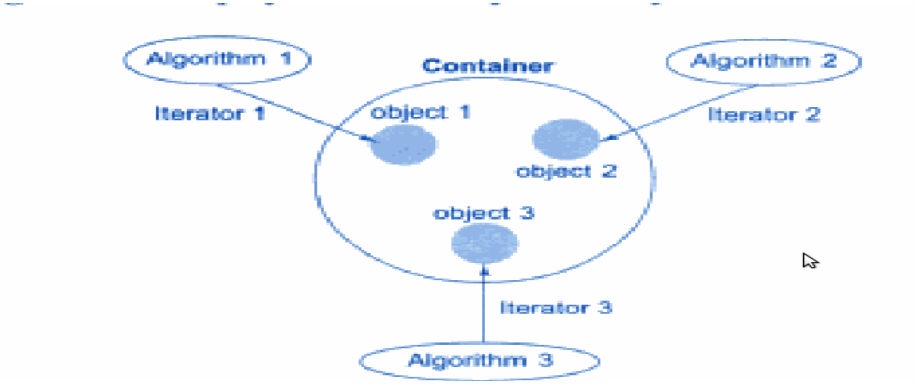
```
class Shape {
  protected:
    int width, height;

  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

STANDARD TEMPLATE LIBRARY: (STL)

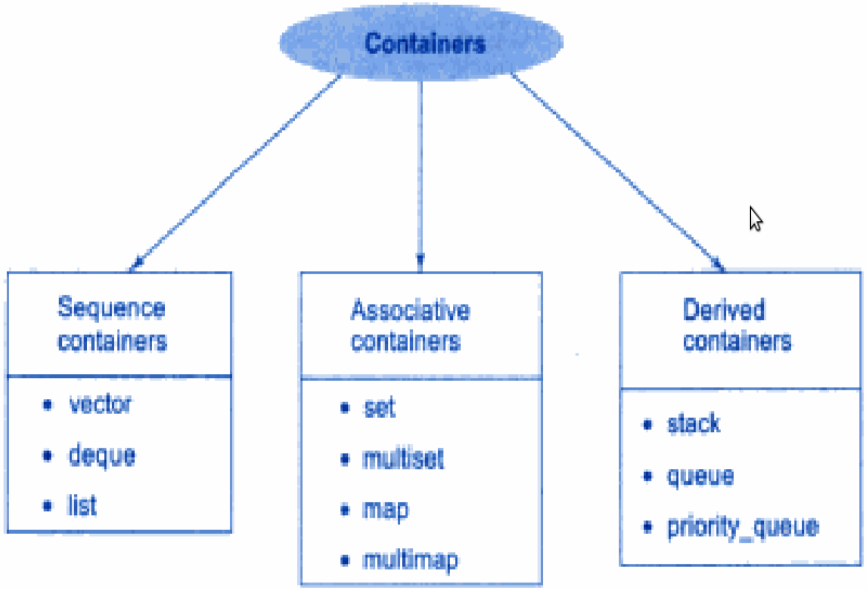
- A set of general-purpose templatized classes and functions used as a standard approach for storing and processing of data. The collection of these generic classes and functions is called the *Standard Template Library.*
- **Components of STL:**
  - The STL contains several components:
    - Containers
    - Algorithms
    - Iterators



- A *container* is an object that actually stores data. It is a way data is organized in memory.
- An *algorithm* is a procedure that is used to process the data contained in the containers.
- An *iterator* is an object that points to an element in a container.

Containers:

- Containers are objects that hold data.
- The STL defines ten containers which are grouped into three categories.



- **Sequence Containers:** Sequence containers store elements in a linear sequence. Each element is related to other elements by its position along the line.
  - The STL provides three types of sequence containers:
    - **Vector:** It is a dynamic array.
    - **List:** It is a bidirectional linear list.
    - **Deque:** A double-ended queue.
- **Associative Containers:** Associative containers are designed to support direct access to elements using keys. They are not sequential.
  - The STL provides four types of associative containers:
    - **Set:** An associative container for storing unique sets.
    - **Multiset:** An associative container for storing non-unique sets.

Both set and multiset can store a number of items and provide operations for manipulating item using the values as the keys.

- **Map:** An associative container for storing unique key/value pairs. Each key is associated with only one value.
- **Multimap:** An associative container for storing key/value pairs in which one key may be associated with more than one value.

- **Derived Containers:** These are also known as container adaptors. They are: stack, queue and priority_queue.
  - **Stack:** A standard stack. Last-in-First-Out (LIFO)
  - **Queue:** A standard queue. First-In-First-Out (FIFO)
  - **Priority-queue:** A priority queue. The first element out is always the highest priority elements.

Algorithms:

Algorithms are functions that can be used generally across a variety of containers for processing their contents.

STL provides more than sixty standard algorithms.

STL algorithms can be classified as:

- Retrieve or nonmutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

Iterators:

Iterators behave like pointers and are used to access container elements.

They are often used to traverse from one element to another, a process known as iterating through the container.

Types:

- Input
- Output
- Forward
- Bidirectional
- Random

The input and output iterators are used to traverse in a container.

The forward iterator supports all operations of input and output iterators and also retains its position in the container.

A bidirectional iterator provides the ability to move in the backward direction in the container.

A random access iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location.

****

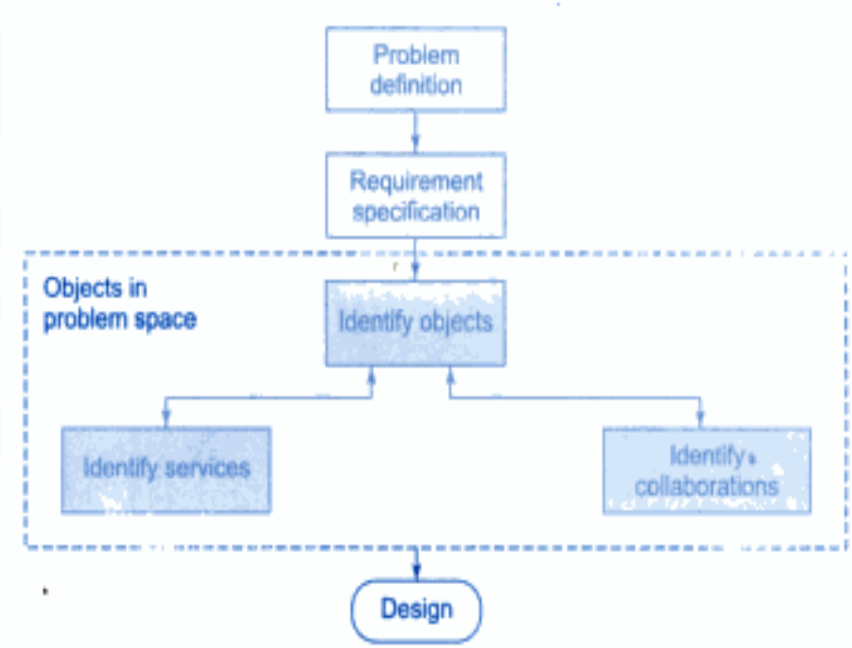
STEPS IN OBJECT-ORIENTED ANALYSIS:

Object-oriented analysis provides a simple, powerful, mechanism for identifying objects, the building blocks of the software to be developed.

The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system function.

The object-oriented analysis (OOA) approach consists of the following steps:

1. Understanding the problem.
2. Drawing the specifications of requirements of the user and the software
3. Identifying the objects and their attributes.
4. Identifying the services that each object is expected to provide.
5. Establishing inter-connections between the objects in terms of services required and services rendered.



Problem Understanding: The first step in the analysis process is to understand the problem of the user. The problem statement should be refined and redefined.

This will enable the software engineers to have a highly focused attention on the solution of the problem.

Requirements Specification: Based on the user requirements, the specifications for the software should be drawn. The developer should state clearly

- What outputs are required?
- What processes are involved to produce these outputs?
- What inputs are necessary?
- What resources are required?

Identification of Objects: Objects can be identified in terms of the real-world objects as well as the abstract objects. The application may be analyzed by using one of the following two approaches.

- **Data flow diagram (DFD):**
  - The application can be represented in the form of a data flow diagram indicating how the data moves from one point to another in the system. It is also known as *data flow graph* or a *bubble chart.*
- **Textual analysis (TA):**
  - This approach is based on the textual description of the problem or proposed solution. The nouns are good indicators of the objects. The names can further be classified as proper nouns, common nouns, and mass or abstract nouns.

The fundamentals of data flow diagram is shown below:

Identification of Services: Once the objects in the solution space have been identified, the next step is to identify a set of services that each object should offer.

Establishing Interconnections: This step identifies the services that objects provide and receive. The information flow diagram (IFD) or an entity-relationship (ER) diagram may be used to enlist the information.

STEPS IN OBJECT ORIENTED DESIGN:

The object oriented design (OOD) approach may involve the following steps:

1. Review of objects created in the analysis phase.
2. Specification of class dependencies.
3. Organization of class hierarchies.
4. Design of classes.
5. Design of member functions
6. Design of driver program

Review of Problem Space Objects:

The main objective of this review exercise is to refine the objects in terms of their attributes and operations and to identify other objects that are solution specific.

Class Dependencies:

Analysis of relationships between the classes is central to the structure of a system. It is important to identify appropriate classes to represent the objects in the solution space and establish their relationships.

The major relationships that are important in the context of design are:

1. Inheritance relationships
2. Containment relationships
3. Use relationships

- **Inheritance relationship** is the highest relationship that can be represented in C++. It is a powerful way of representing a hierarchical relationship directly.

- **Containment relationship** means the use of an object of a class as a member of another class. This is an alternative and complimentary technique to use the class inheritance.
- **Use relationship** gives information such as the various classes a class uses and the way it uses them.
  - **Ex:**
    - A reads a member of B
    - A calls to member of C

Organization of Class Hierarchies:

Organization of the class hierarchies involves identification of common attributes and functions among a group of related classes and then combining them to form a new class. The new class will serve as the super class and the others as subordinate classes.

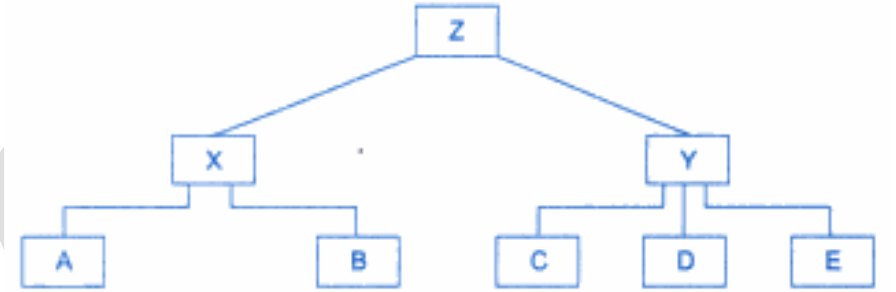
Design of Classes:

We have identified classes, their attributes, and set of operations required by the concept a class is representing. The important issue is to decide what functions are to be provided. For a class to be useful, it must contain the following functions, in addition to the service functions.

1. Class management functions
   a. How an object is created?
   b. How an object is destroyed?
2. Class implementation functions.
   What operations are performed on the data type of the class?
3. Class access functions.
   How do we get information about the internal variables of the class?
4. Class utility functions.
   How do we handle errors?

Design of Member functions:

1. Classes and objects
2. Data members
3. interfaces
4. dependencies
5. class hierarchy



(c) Second level of hierarchy

Design of the Driver Program:

The driver program is mainly responsible for:

1. receiving data values from the user
2. creating objects from the class definitions
3. arranging communication between the objects as a sequence of messages for invoking the member functions
4. displaying output results in the form required by the user

MANIPULATING STRINGS:

A string is a sequence of characters.

The string class is very large and including many constructors, member functions and operations.

Constructors:

String( ); //For creating an empty string

String(const char *str); // For creating string object from a null-terminated string

String(const string & str); // For creating a string object from other string object.

INPUT OUTPUT STREAMS:

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device likes a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device likes a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

There are following header files important to C++ programs

<iostream> This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.

<iomanip> This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.

<fstream> This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main() {
  char str[] = "Hello C++";
  cout << "Value of str is : " << str << endl;
}
```

The Standard Input Stream (cin)

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
  char name[50];

  cout << "Please enter your name: ";
  cin >> name;
  cout << "Your name is: " << name << endl;

}
```

The Standard Error Stream (cerr)

The predefined object cerr is an instance of ostream class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object cerr is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The cerr is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
  char str[] = "Unable to read....";

  cerr << "Error message : " << str << endl;
```

```
}
```

C++ EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- throw − A program throws an exception when a problem shows up. This is done using a throw keyword.
- catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

****