PANIMALAR INSTITUTE OF TECHNOLOGY

(JAISAKTHI EDUCATIONAL TRUST)

CHENNAI 600 123

DEPARTMENT OF CSE

EC6301 OBJECT ORIENTED PROGRAMMING AND DATA STRUCTURES

II YEAR – III SEMESTER – ECE

LECTURE NOTES

UNIT – I

UNIT I

DATA ABSTRACTION & OVERLOADING

Overview of C++ – Structures – Class Scope and Accessing Class Members – Reference Variables – Initialization – Constructors – Destructors – Member Functions and Classes – Friend Function – Dynamic Memory Allocation – Static Class Members – Container Classes and Integrators – Proxy Classes – Overloading: Function overloading and Operator Overloading.

INTRODUCTION

Object-Oriented programming (OOP) has become the preferred programming approach by the software industries. It offers a powerful way to cope up with the complexity of real-world problems. Among the OOP languages available today, C++ is by far the most widely used language.

BASICS OF C++ ENVIRONMENT

C++ is an object oriented programming language. It was developed by **Bjarne Strousturp at AT&T Bell Laboratories** in Murray Hill, New Jersey, USA, in the early 1980's. C++ is a superset of C.

Every C++ program must have a **main( )**.

Like C, the statements **terminate with semicolons**.

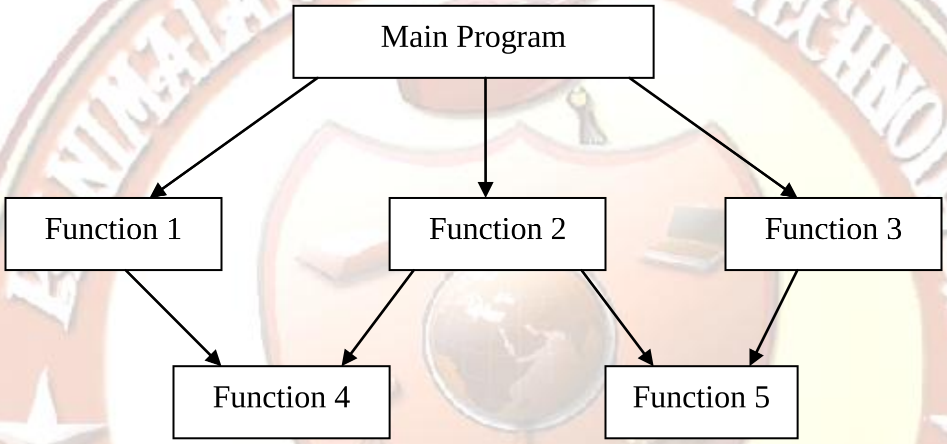
File Extension:

C++ programs have the file extension **".CPP"**

Structured Programming:

- A structured program is built by breaking down the program's primary purpose into smaller pieces that then become functions within the program.

- Each function can have its own data and logic.
- Information is passed between functions using parameters and functions can have local data that cannot be accessed outside the function scope.
- It helps you to write cleaner code and maintain control over each function.
- By isolating the processes within the functions, a structured program minimizes the chance that one procedure will affect another.



Limitations:

- Data is given second class status in the procedural paradigm even though data is the reason for program’s existence.
- Data types are processed in many functions, and when changes occur in data types, modifications must be made at every location that acts on those data types within the program.
- Structured programming components – functions and data structures doesn’t model the real world very well.
- It emphasizes on fitting a problem to the procedural approach of a language.

<u>The iostream File:</u>

#include <iostream.h>

The header file **iostream.h** should be included at the beginning of all the programs that use input/output statements.

<u>Output Operator:</u>

cout << " Hello World";

The identifier cout (pronounced as 'C Out') is a predefined object that represents the standard output stream (screen). The operator << is called the **insertion or put to operator**.

<u>Input Operator:</u>

cin >> variable;

The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream (keyboard). The operator >> is known as extraction or get from operator.

<u>Cascading of I/O Operators:</u>

The statement

cout << " Sum = " << sum <<"\n";

first sends the string "Sum = " to **cout** and then sends the value of sum. Finally, it sends the newline character \n so that the next output will be in the new line. Similarly the input operator can be cascaded as follows:

cin >> num1 >> num2;

The values are assigned from left to right. That is if we type 27 and 13 then 27 will be assigned to num1 and 13 will be assigned to num2.

<u>Tokens:</u>

The smallest individual units in a program are known as tokens. C++ has the following tokens:

- **Keywords**
- **Identifier**

- **Constants**
- **Strings**
- **Operators**

<u>Keywords:</u>

C++ has its own keywords in addition to the keywords in C. Some of them are

catch	new	template
class	operator	this
delete	private	throw
friend	protected	try
inline	public	virtual

<u>Identifiers:</u>

Identifiers refer to the name of the variables, functions, arrays, and classes etc., created by the programmer. The rules for forming the variables are

- Only alphabetic characters, digits and underscores are permitted
- The name cannot start with the digit
- Uppercase and lowercase letters are distinct
- A declared keyword cannot be used as a variable name

C++ allows the declaration of a variable anywhere in the scope. It makes the program easier to write and reduces the errors also makes the program easier to understand because the variables are declared in the context their use.

<u>CONSTANTS:</u>

Constants refer to the fixed values that do no change during the execution of a program.

Like C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings.

(ex) 765, 34.56, ’D’, ”LAB” .

DATA TYPES:

User defined type

- structure
- union
- class
- enumeration

Derived Type:

- Array
- Function
- Pointer
- Reference

Built –in type

- Integral type
  - int
  - char
- void
- Floating type
  - float
  - double

USER DEFINED DATA TYPES:

STRUCTURE:

- C Structure is a collection of different data types which are grouped together and each element in a C structure is called member or structure is a heterogeneous data structure which can store data elements of mixed data types. Keyword struct is used for creating a structure.

SYNTAX:

struct structure_name

```
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
};
```

STRUCTURE VARIABLE DECLARATION

When a structure is defined, it creates a user-defined type but, no storage is allocated. For the above structure of person, variable can be declared as:

Example :

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

Inside main function:

struct person p1, p2, p[20];

Another way of creating sturcture variable is:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
}p1 ,p2 ,p[20];
```

In both cases, 2 variables *p1*, *p2* and array *p* having 20 elements of type **struct person** are created.

CLASS:

- Class is a user defined data type which wraps both functions and variables as a single unit.
- The functions inside the class are called as member functions and the variables inside the class are called as member data. In general they are commonly called as members of the class.

SYNTAX:

```
class <class-name>
{
Access specifier:     //private or public or protected
Data Members;
Access specifier:
Member Functions;
};
```

EXAMPLE

```
#include <iostream.h>
#include <conio.h>
class marks
{
int m;
int roll;
public :
void get()
{
cout <<"\n Enter the roll no  and  mark\n";
cin>>roll>>m;
}
void show()
{
cout <<"\n The roll no is\n"<<roll;
cout <<"\n The mark is \n"<<m;
}};
void main()
{
mark a;
a.get();
a.show();
}
```

- In the above program ‘mark’ is the name of the class.
- It contains 2 member data’s ‘m’ and ‘roll’
- It also contains 2 member functions ‘get()’ and ‘show()’
- In the main( ) an object ‘a’ is created and it is used to accessed the member functions which will access the member data’s.

DERIVED DATA TYPES:

ARRAY:

Array is a homogenous data structure which can store ‘n’ elements of same data type.

SYNTAX:

Data-type arr-name[index];

- Data-type specifies the data type of the elements that the array is going to hold.
- arr-name, specifies the name of the array.
- Index specifies the maximum number of the elements that the array is going to hold.

EXAMPLE:

```
int a[3]={1,2,3};
```

POINTERS:

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a char as to point to an int or a float.

The declaration of pointers follows this format:

```
type * name;
```

where, type is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! But the type of the data the pointer points to. For example:

```
int * number;
char * character;
float * greatnumber;
```

POINTERS AND ARRAYS:

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept.

For example, supposing these two declarations:

int numbers [20];

int * p;

POINTER INITIALIZATION

When declaring pointers we may want to explicitly specify which variable we want them to point to:

int number;

int *tommy = &number;

OPERATORS IN C++:

::	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:

Suppose there are two variables with the same name, one which is defined globally another is declared locally inside a function. If we attempt to access the variable a, we always access the local variable as the rule says, the local variable gets more priority over global variable. C++ provides a way to access the global variable using scope resolution operator.

Example:

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

```
int a = 27;
void main()
  {
          int a = 13;
          cout << "\nLocal value " = << a <<" Global Value = "<< ::a
          ::a = 30;
          cout << "\nLocal value " = << a <<" Global Value = "<< ::a
  }
```

Output:

```
 Local value = 13   Global Value = 27
 Local value = 13   Global Value = 30
```

<u>TYPECASTING:</u>

If we carry out the operation between an int and float the int is promoted to a float before performing the operation. This conversion takes place automatically. As against this, data conversions can be carried out by the programmer explicitly. Two different types of typecasting are supported in C++.

Example:

```
          int total = 450 , n = 7
          float avg;
          avg = total /n;
          cout << "average = " << avg;
```

Output:

```
          average = 64
```

After typecasting:

```
          int total = 450 , n = 7
```

```
float avg;
avg = (float)total /n;        // C style
cout << “average = “ << avg;
```

Output:

```
average = 64.29
```

C++ style:

```
avg  = float(total)/n;
```

<u>EXPRESSIONS:</u>

An expression is a combination of operators, constants and variables arranged as per the rule of the language. An expression may consist of one or more operands, and zero or more operators to produce a value.

- **<u>Constant Expressions</u>**

  Constant expressions consist of only constant values.

  (Ex)   45

  90+10

  ’x’

- **<u>Integral Expressions</u>**

  Integral expressions are those which produce integer results.

  (Ex)

  m

  m * n

  9 + int(29.0)

- **<u>Float Expressions</u>**

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

  (Ex)

  x + y

7 + float (56)

- **Pointer Expressions**

  Pointer expression produces address values.

  (Ex) &m

  ptr

  ptr + 1

- **Relational Expressions**

  Relational expression yield results of type bool which takes a value true or false.

  (Ex)

  x < = z

  e + b = = v + g

  m + n > 100

- **Logical Expressions**

  Logical expressions combine two or more relational expressions and produces bool type results.

  (Ex)

  a>b && c= =34

  x = = 20 || y= 8

- **Bitwise Expressions**

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

  (Ex)

  x << 3

  y >> 1

Explain the control structures of C++ with suitable examples. ***(NOV/DEC 2010)(MAY/JUNE 2013)***

CONTROL STRUCTURES:

- **if…else** statement
- **switch** statement
- **do while** statement
- **while** statement
- **for** statement

Conditional structure: if and else

The if + else structures can be concatenated with the intention of verifying a range of values.

The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
cout << "x is positive";
else if (x < 0)
cout << "x is negative";
else
cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

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

}

and its functionality is simply to repeat statement while the condition set in expression is true.

The do-while loop

Its format is:

do statement while (condition);

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the Following example program echoes any number you enter until you enter 0.

The for loop

Its format is:

for (initialization; condition; increase) statement;

It works in the following way:

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. Condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream.h>
```

```
int main ()
{
for (int n=10; n>0; n--)
{
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```

The switch statement

This is a multiple- branching statement where, based on a condition, the control is transferred to one of the many possible points.

Syntax:

```
switch(expression)
{
        case 1:
        {
        action1;
        }
        case 2:
        {
        action2;
        }
        case 3:
        {
        action3;
```

```
                }
                default:
                {
                action4;
                }
        }
        action5;
```

FUNCTIONS IN C++:

Syntax :

```
Return type function name ();    //Empty function
Return type function name (arg);        // Function with argument
```

Example.

```
void show();                         //Function declaration
main()

{
        …..
        show();                      //Function call
        …….
}
void show()                          //Function definition
{
        ………
        ………                          //Function body
}
```

- ❖ In C, main() does not return any value.
- ❖ The main () returns a value of type int to the operating system.

```
int main();
int main(int a, int b);
```

❖ The function that has a return value should use the return statements for termination.

```
int main()
{
  ……
  return 0;
}
```

PROGRAM USING CALL BY VALUE, CALL BY ADDRESS AND CALL BY REFERENCE:

```
#include<iostream.h>
//prototype declarations
void swapv(int, int)                    // call by value
void swapa(int *, int *)    // call by address
void swapr(int &, int &)  // call by reference
void main ()
{
  int a =10,b =20;
  swapv(a,b);
  cout << " Using Call by value\n" << "a =  " << a <<" b = " << b;
  swapa(&a,&b);
  cout << "\nUsing Call by address \n " << "a = " << a << " b = " << b;
  swap(a,b);
  cout << "\n Using Call by reference \n" << "a = " << a << " b = " << b;
  }
  void swapv(int x, int y)
  {
        int t;
        t = x;
        x = y;
        y = t;
  }
 void swapa( int *v, int *p)
  {
        int t;
        t = * v;
        *v = *p;
```

```
          *p = t;
  }
void swapr(int &r, int &v)
{
          int t;
          t = r;
          r = v;
          v = t;
}
```

Output:

```
Using Call by value
 a = 10     b = 20
Using Call by address
a = 20      b = 10
Using Call by reference
a = 10      b = 20
```

INLINE FUNCTIONS :

An inline function is a function that is expanded in line when it is invoked. That is, the complier replaces the function call with the corresponding function code. The inline functions are defined as follows:

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

Example :

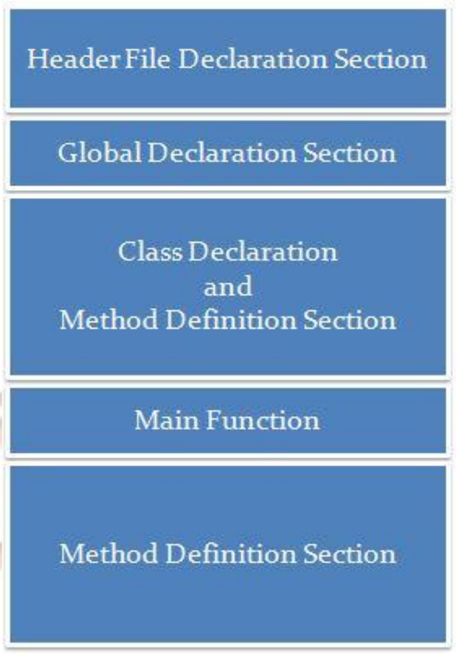
```
         inline int  cube(int a)
          {
              return (a*a*a);
```

```
}
```

Program :

```
#include<iostream.h>
  inline float mul(float x, float y)
       {
                return (x*y);
       }
inline float div(double p, double q)
       {
                return( p / q )
       }
 int main() {
          float a=12.35;
          float b=9.32;
          cout<< mul(a,b);
          cout<<div(a,b);
          return 0;
       }
```

STRUCTURE OF C++ PROGRAM:


Header File Declaration Section
Global Declaration Section
Class Declaration and Method Definition Section
Main Function
Method Definition Section

Section1: Header File Declaration Section

- Header files used in the program are listed here.
- Header File provides **Prototype declaration** for different library functions.
- We can also include **user define header file**.
- Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

Global Variables are declared here.

Global Declaration may include -

Declaring Structure

Declaring Class

Declaring Variable

Section 3 : Class Declaration Section

Actually this section can be considered as sub section for the global declaration section.

Class déclaration and all methods of that class are defined here.

Section 4 : Main Function

Each and every C++ program always starts with main function. This is entry point for all the function. Each and every method is called indirectly through main. We can create class objects in the main.

Operating system call this function automatically.

Section 5 : Method Definition Section

This is optional section. Generally this method was used in C Programming.

<u>OBJECT ORIENTED PROGRAMMING (OOP) CONCEPTS :</u>

What is object oriented paradigm?

<u>Definition of OOP:</u>

It 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.

OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.

OOP allows decomposition of a problem into a number of entities called *objects* and then builds data and functions around these objects.

The organization of data and functions in object-oriented programs is shown in fig.1.1. The data of an object can be accessed only by the functions associated with that object. However functions of one object can access the functions of other objects.

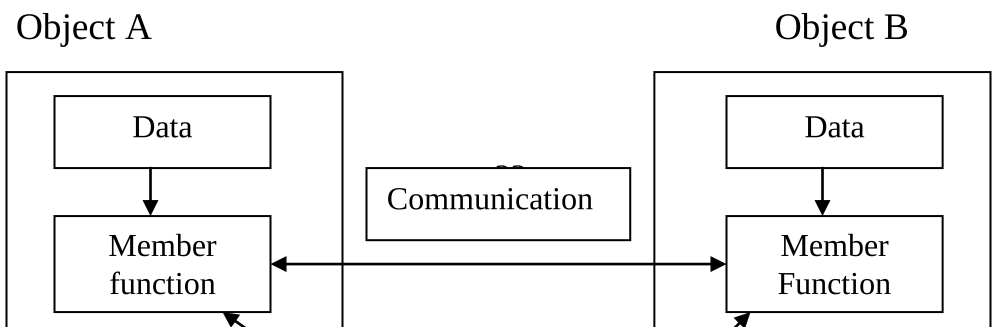


Fig. 1.1 Organization of data and functions in OOP

- *Compare and contrast Structured Programming and Object Oriented Programming.**(NOV/DEC 2010)***

Comparison between Structured and Object Oriented Programming:

Structured Programming	**Object Oriented programming**
Emphasis is on doing things	Emphasis is on data
Large programs are divided into smaller programs known as functions	Programs are divided into what are known as objects.
Most of the functions share global data	Functions that operate on data of the object are tied together in the data structures
Data move openly around the system from function to function.	Data is hidden and cannot be accessed by external function.
Its components doesn't model the real world objects	Its components model the real world objects
Employs *top-down* approach in	Follows *bottom-up* approach in

program design.	program design.

- *Explain the following concepts of object oriented programming in detail with example.* ***(NOV / DEC 2007)***
  - *(i). Data abstraction*
  - *(ii). Inheritance*
  - *(iii). Polymorphism*
  - *(iv). Object*
- *Distinguish between Data Encapsulation and Data Abstraction.* ***(NOV/DEC 2010)***

BASIC CONCEPTS OF OBJECT ORIENTED PROGRAMMING:

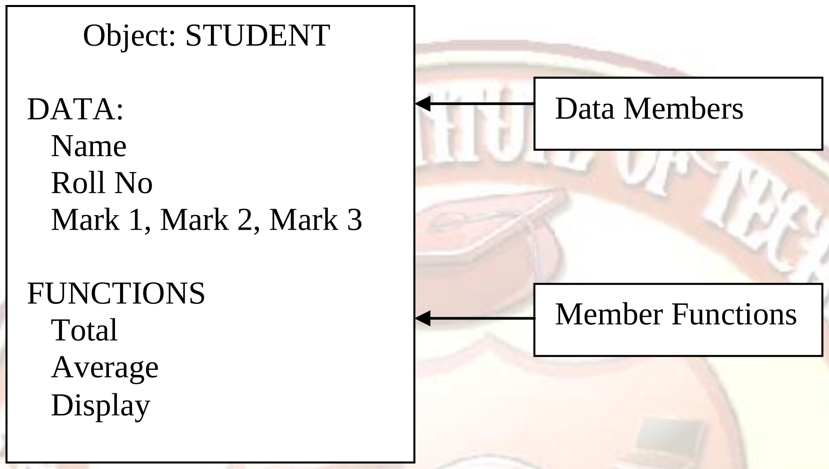
These include:

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

OBJECTS:

Objects are the *basic run-time entities* in an object oriented programming. They may represent a person, a place, a bank account or any item that the program has to handle. They may represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. **Objects are the instances of the classes.**

When a program is executed, the objects interact by sending messages to another. Each object contains data and code to manipulate the data. Objects can interact without having to know the details of each others data or code. It is sufficient to know the type of message accepted, and the type of message accepted and the type of response returned by the objects. Fig.1.2. shows the representation of an object.



CLASSES:

The entire set of data and code of an object can made a user defined data type with the help of a class. In fact the objects are variable of the type class. Once class has been defined we can create any number of objects belonging to that class. Class is similar to structure in c. In short, **a class serves as a blueprint or a plan or a template.** It specifies what data and what functions will be included in objects of that class. **A class is thus a collection of objects of similar type. Classes are user-defined data type and behave like the built in types of a programming language.**

For example

mango, apple and orange are member of the class fruit.

DATA ABSTRACTION AND ENCAPSULATION:

The wrapping up of data and functions into a single unit is known as **encapsulation.** It is the most striking feature of the class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide interface b/n the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

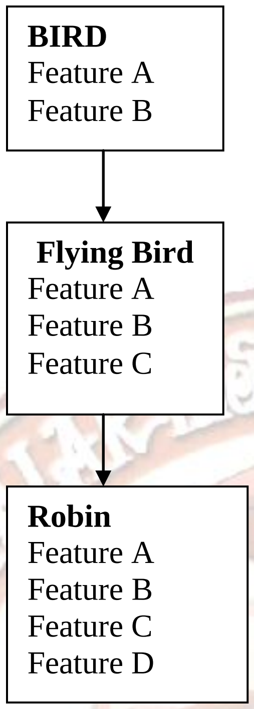
Abstraction represents the act of representing the essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes. The attributes are called data members and functions are called member functions or methods.

Since the classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT).**

INHERITANCE:

It 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 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'.

This concept provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by a deriving a new class from an existing one. The new class will have the combined features of both the classes.



<u>POLYMORPHISM:</u>

It means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example the operation addition will generate sum if the operands are numbers whereas if the operands are strings then the operation would produce a third string by concatenation.

The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading.**

A single function name can be used to handle different types of tasks based on the number and types of arguments. This is known as **function overloading.**

<u>DYNAMIC BINDING:</u>

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 the code associated with the given procedure call is not known until the time of the call at run-time. It is associated with the polymorphism and inheritance.

MESSAGE PASSING:

The process of programming in OOP involves the following basic steps:

- Creating classes that define objects and their behavior
- Creating objects from class definitions
- Establishing communication among objects

A *message* for an object is request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result.

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

E.g.: employee.salary(name);

Object: employee

Message: salary

Information: name

Benefits of OOP:

(*Describe the advantages and applications of OOPs technology. (May/June 2007))*

- Through inheritance, we can eliminate redundant code and extend the use of existing classes
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is easy to partition the work in a project based on the objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

- Object oriented systems can be easily upgraded from small to large systems.
- It is possible to map multiple instances of an object to co- exits without any interference.
- Message passing techniques for communication between objects make the interface description with external systems much simpler.

Applications of OOP:

- Development of Operating Systems
- Creation of DLL (Dynamic Linking Library)
- Computer Graphics
- Network Programming (Routers, firewalls)
- Voice Communication (Voice over IP)
- Web-servers (search engines)

CLASS SCOPE AND ACCESSING CLASS MEMBERS:

Definition of a class:

A class is a way to bind the data and its associated functions together. It allows the data to be hidden if necessary from external use. While defining a class, we are creating a new *abstract data type* that can be treated as a built-in data type. A class specification has two parts:

1. Class declaration – describes the type & scope of its members
2. Class function definitions – describe how the class functions are implemented.

Class Scope:

A class data member and member functions belong to that class's scope. Non- member functions are defined at file's scope.

The variables declared inside the class are known as **data members.**

The functions declared inside the class are known as **member function.**

General form of class declaration

```
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 the 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 keywords **private and public** are known as visibility labels and it specify which members are private which of them are public. These should followed by a colon.

Private members can be accessed only within the class whereas public members can be accessed from outside the class. **By default, the members of a class are private**. If both the labels are missing, the members are private to the class.

Access Specifiers :

Access specifiers are used to access the data members and member functions of the class. There are three types of access specifiers:

- private
- public
- protected

private: the attributes that provide services (access) only to member functions are kept to be private.

What is the default access mode for class members? (Nov/Dec 2007)

private is the default access mode for class members.

public: The attributes that provide services (access) to outsiders are to be kept as public.

protected: A member declared as protected is accessible by the member functions within its class and class immediately derived from it. It cannot be accessed by functions outside these two classes (base and derived).

When a protected member function is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance.

A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance.

```
class alpha
{
        private:                // optional
                …….             // visible to member functions
                …….             // within its class

        protected:
                …….             // visible to member functions
                …….             // of its own and derived class

        public:
                …….             // visible to all functions
        …….         // in the program
}
```

<u>Private Member Functions</u>

Usually the member functions are normally placed in the public part, but in some situations some functions like deleting an account in the customer are more vulnerable, so it will be placed inside the private part. **A private member function can only be called by another function that is a member of its class.** Even objects invoke a private function using dot operator.

Example:

```
class max
{
        int v, p;
        void in( )
        {
                cout << “Enter two numbers “;
                cin >> v >> p;
        }
  public:
        void calc()
        {
                in();   //simple call no object is used
                if ( v>=p)     cout << “ Maximum = “<<v;
                else
                        cout << “Maximum = “<< p;
        }
};
void main()
{
        max A;
        A.in();         //won’t  work
        A.calc();
}
```

Inheritance and Accessibility:

Access Specifier	**Accessible from Own Class**	**Accessible from Derived Class**	**Accessible from Objects Outside Class**
public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Arrays within classes:

The arrays can be used as member variables in a class.

```
class array
{
        int a[10];
   public:
        void  input();
        void out();
};
```

The array variable a[] declared as a private member of the class array can be used in the member functions, like any other variable. We can perform any operations on it.

Arrays of Objects:

```
class student
{
        int rno;
        char name[10];
        int total;
  public:
        void in();
        void out();
};
void student :: in()
        {
                cout << “Enter two numbers “ ;
                cin >> v >> p;
        }
void student:: out()
        {
                // calling member function

                cout << v<<p;
        }
void main()
{
        student cse[10];
        for(int i =0;i<10;i++)
                cse[i].in();
        cout<<”MarkList of CSE \n“;
        for(i=0;i<10;i++)
                cse[i].out();
}
```

The identifier **student** is a user defined data type and can be used to create objects that relate to different categories of the students. The array **cse** contains 10 different objects namely **cse[0], cse[1], cse[2]** and so on.

Objects as function arguments:

We can use object as function argument in two ways

1. Copy of the entire object is passed to the function (pass by value)

2. Only the address of the object is transferred to the function (pass by reference).

REFERENCE VARIABLES:

A reference variable provides an alias (alternate name) for the previously defined variable.

If we make the variable SUM a reference to the variable TOTAL, then SUM and TOTAL can be used interchangeably to represent that variable.

- Both the variables refer to the same data object in the memory.
- A reference must always refer to something. NULLs are not allowed.
- Reference variable must be initialized at the time of declaration.
- Once initialized, it cannot be changed to another variable.

SYNTAX:

Data type &ref-var-name = variable-name;

Example :

```
int y=10;
int &x=y;
```

- In the above code 'y' is normally declared and 'x' is an alias for y.
- This means that both x and y share the same memory locations and any modifications made to any of them will reflect both the variables

Example:

```
int main()
{
int a=9;
int &aref=a;
a++;
cout<<"a="<<a;
cout<<Aref="<<aref;
return 0;
}
```

- A reference variable must be initialized at the time of declaration.
- Initialization of a variable is completely different from assignment to it.

```
#include <iostream.h>
void main ()
{
  // declare simple variables
  int    i;
  double d;
  // declare reference variables
  int&    r = i;
  double& s = d;
  i = 5;
  cout << "Value of i : " << i << endl;
  cout << "Value of i reference : " << r  << endl;
  d = 11.7;
  cout << "Value of d : " << d << endl;
  cout << "Value of d reference : " << s  << endl;
  return 0;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

CONSTRUCTORS AND DESTRUCTORS:

Constructor:

A constructor is a special member function whose task is to **initialize the objects of its class** when it is created. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. This is known as automatic initialization of objects.

Syntax:

    <class_name>(arguments);

A constructor is declared and defined as follows:

```
class sample
{
        int m,n;
public:
        sample()
        {
                m = n = 0;
        }
        // some code
};
void main()
{
        sample s;     // object is created
        // some code
}
```

During the object creation, it also initializes the data member’s m and n to 0.

Characteristics of a constructor:

- They should be declared in the public section
- They are invoked automatically when the objects are created
- They do not have return types, not even void so they cannot return values.
- They cannot be inherited

- They can have default arguments
- It cannot be virtual
- We cannot refer to their addresses
- An object with a constructor or destructor cannot be used as a member of a union
- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.

Types of constructor:

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Dynamic Constructor

DEFAULT CONSTRUCTOR:

A constructor that **accepts no parameters** is called the default constructor.

Example:

```
class sample
{
private:
   int a,b;
public:
  sample();   //default constructor(class name and function name are same. Constructor has no arguments)
.….
};
sample::sample()
{
```

```
                a=0;
                b=0;
        }
```

PARAMETERIZED CONSTRUCTORS:

The constructors that **can take arguments** are called parameterized constructors. By using this we can initialize different objects to different values when they are created.

We must pass the initial values as arguments to the constructor function when an object is declared.

This can be done in two ways

- By calling constructor explicitly(Explicit call)
- By calling constructor implicitly(Implicit call)

Example:

Inside the class definition:

```
sample (int a, int b)
{
        m = a;
        n = b;
}
```

Inside the main() :

sample s (27,30);   // implicit call – easier to implement

sample s = sample(14,25)//explicit call—the statement creates an object s and passes the values 14 and 25 to it.

COPY CONSTRUCTOR:

A copy constructor takes **a reference to an object of the same class as itself as an argument.**

The main use of copy constructor is to initialize the object while in creation, also used to copy an object. This copy constructor allows the programmer to create a new object from an existing one by initialization

Example:

It sets the value of every data element of **s3** to the value of the corresponding data element of **s2**

<u>Constructors with default arguments:</u>

It is possible to define constructors with default arguments.

Example:

Inside the class definition:

sample(int a, int b = 10) // The default value of the argument b is 10.

Inside the main():

sample s2(20);

assigns 20 to x and 10 to y.

where as

sample s5(25,35) ; //assigns 25 to x and 35 to y

<u>MULTIPLE CONSTRUCTORS IN A CLASS:</u>

C++ permits to use the **default constructor and parameterized constructors in the same class.**

Example:

```
#include<iostream.h>
class sample
{
        int x,y;
public:
        sample()      // Default Constructor
        {
                x = y = 0;
        }
        sample(int a, int b) // Parameterized Constructor
        {
                x = a;
                y = b;
        }
        sample (sample &s)        // copy constructor
        {
                x = s.x;
                y = s.y;
        }
};
void main()
{
        sample s1;   // invokes default constructor

        sample s2(10,20);  //invokes parameterized constructor

        sample s3 = s2;     //invokes copy constructor

        sample s4(s3);                //invokes copy constructor

}
```

Dynamic constructors:

The constructors can also be used to **allocate memory while creating objects**. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. The memory is allocated with the help of new operator.

Example:

```
        class sample
        {
                int *ptr;
        public:
                sample(int x)
                {
                        ptr = new int[x];
                }
                // some code
        };
        void main()
        {
                cout<< "Enter the size of the array"
                int n;
                cin >> n;
                sample s(n);
}
```

DESTRUCTORS:

A destructor is used to **destroy the objects that have been created by a constructor**. The **name of the destructor is same as the class name but preceded by a tilde (~)**. A destructor never takes any argument nor does it return any value. It will be invoked by the compiler upon the exit from the program.

- Whenever a new operator is used to allocate memory in the constructors, we should use delete to free that memory.

Example: Program to demonstrate destructor

```
class A
{
        A( )
         {
                cout << "Constructor called";
          }

        ~A( )
                {
                        cout << "Destructor called";
                }
};

void main()
{
        A obj1;   // Constructor Called
```

```
            int x=1;
               if(x)
                  {
                     A obj2;  // Constructor Called
                   }   // Destructor Called for obj2
    } //  Destructor called for obj1
```

Example Program for mark details which demonstrate constructor and destructor

```
# include <iostream.h>
# include <conio.h>
class percent
{
float total_marks;
float max_marks;
public:
percent(float tm, float mm)
{
total_marks = tm;
max_marks = mm;
}
percent()
{
cout<<"Enter Marks Obtained : ";
cin>>total_marks;
cout<<"Enter Max. Marks : ";
cin>>max_marks;
}
percent(percent &p)
{
total_marks = p.total_marks;
max_marks = p.max_marks;
}
~percent()
{
cout<<endl;
cout<<endl<<"Marks Obtained : "<<total_marks;
cout<<endl<<"Out Of : "<<max_marks;
cout<<endl<<"Percentage Marks : "<<(total_marks*100)/max_marks;
}
};
void main()
{
        clrscr();
        percent a;
        percent b(432,450);
        percent c(a);
}
```

FRIEND FUNCTIONS :

In OOP environment, the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. To allow a function (non-member) to have access to the private data of a class, the function has to be made friendly to that class. To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below :

```
class ABC
{
                …….                          // data members
                …….
        public :
                …….                          // member functions
                …….
                friend void xyz(void);       // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope resolution operator **: :**.

The functions that are declared with the keyword friend are known as friend functions. A friend function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

Characteristics (certain special characteristics) of Friend functions:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member access.(e.g. A . x).
- It can be declared either in the public or private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

// program to demonstrates the use of a friend function

```
#include<iostream>
class sample
{
        int a;
          int b;
        public:
void setvalue( )
{
        a = 25;
        b = 40;
}

friend float mean (sample s);          // declaration
};
```

```
float mean (sample s)
{
    return float(s.a + s.b) / 2.0 ;
}
int main( )
{
        sample X;                    // object X
        X.setvalue( );
        cout << "Mean Value = " << mean(X) << "\n";
        return 0;
}
```

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

Mean Value = 32.5

The friend function accesses the class variables a and b by using the dot operator and the object passed to it. The function calls mean(**X)** passes the object X by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
        class X
        {
                …….
                …….
                int fun1( );                 // member function of X
                …….
        };
        class Y
        {
                …….
                …….
                friend int X : : fun1( );
                …….
        };
```

The function **fun1 ( )** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
        …….
        friend class X;         // all member functions of X are friends to Z
};
```

//program with a function friendly to two classes

```
#include<iostream.h>
class ABC ;                         // forward declaration
class XYZ
{
                int x;
        public :
                void setvalue(int i) { x = i ;}
                friend void max(XYZ, ABC);
};
class ABC
{
        int a;
        public :
                void setvalue(int i) { a = i ;}
                friend void max(XYZ, ABC);
};
```

```
// definition of friend function
void max(XYZ m, ABC n)
{
        if(m.x >=  n.a)
                cout << m.x;
        else
                cout << n.a;
}
int main( )
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);
        return 0;  }
```

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

```
20
```

Note: The function max( ) has arguments from both XYZ and ABC. When the functions max( ) is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

```
        class ABC;
```

This is known as ‘forward’ declaration.

DYNAMIC MEMORY ALLOCATION :

C++ integrates the operators new and delete.

Operators new and new[]

In order to request dynamic memory we use the operator **new**. "**new"** is followed by a data type specifier and if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated.

Its form is:

Pointer-variable = new data-type

Pointer-variable = new data-type [number_of_elements]

The first expression is used to allocate memory to contain one single element of data type . The second one is used to assign a block (an array) of elements of data type, where number_of_elements is an integer value representing the amount of these.

Example 1:

```
int *p ; // p is a pointer of type int
float *q; //q is a pointer of type float
p= new int;
q= new float;
```

The pointer variables p and q holds the new address generated of its appropriate data types.

Example 2:

```
int *p;
p = new int[10];
float *q = new float[10];
```

The pointer variables p and q holds the new address generated of its appropriate data types with the given size 10.

In the given example, memory will be allocated as 10 integer locations in an array with each of size 2 bytes (total bytes allocated is 20). This address is given to the pointer variable p. similar allocation is done for q with its appropriate data type.

OPERATORS DELETE AND DELETE []:

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements. The value passed as Argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

Example:

```
//New and delete operator:
#include <iostream.h>
int main()
{
  int n, *pointer, c;
  cout << "Enter an integer\n";
  cin >> n;
  pointer = new int[n];
  cout << "Enter " << n << " integers\n";
  for ( c = 0 ; c < n ; c++ )
  cin >> pointer[c];
  cout << "Elements entered by you are\n";
  for ( c = 0 ; c < n ; c++ )
    cout << pointer[c] << endl;
  delete[] pointer;
  return 0;
}
```

Advantages of new operator:

- It automatically computes the size of the data object. We need not use the operator sizeof.
- It automatically returns the correct pointer type, so that there is no need to use a type cast.
- It is possible to initialize the object while creating the memory space.
- Like any other operator, new and delete can be overloaded.

STATIC CLASS MEMBERS:

STATIC DATA MEMBERS:

A data member of a class can be qualified as static. The keyword static is used in front of

A static member variable has certain characteristics such as:

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. Only one copy of that member is created for the entire class and is shared by all the objects of the class, no matter how many objects are created.
3. It is visible only within the class but its lifetime is the entire program.

Static variables are normally used to **maintain values common to the entire class**. For example, a static data member can be used as a counter that records the occurrence of all the objects.

The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Example:

```
#include<iostream.h>
class item
{
        static int count;    // static variable declaration
        int num;
   public:
        void in(int x)
        {
                 num = x;
                count++;
        }
        void out()
        {
                cout << "\nCount = '<< count;
        }
}
int item :: count;  //definition of static member
void main()
{
        item v,r;
        v.out();
        r.out();
        v.in(27);
        r.in(30);
        cout<"After reading data ";
        v.out();
        r.out();
}
```

output :

```
count = 0
count = 0
```

After reading data

```
count = 2
count = 2
```

- while defining static variable, some initial value can also be assigned to the variable.

For example,

```
        int item :: count = 10;
```

STATIC MEMBER FUNCTIONS:

A member function is declared static has the following properties:

- A static function can have access to only other static member functions or variables declared in the same class.
- A static member function can be called using the class name instead of object as follows:

  class name :: function name;

Example Program:

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

```
class test
{
int code;
static int count;  //static member variable
public:
void setcode(void)
{
code=++count;
}
void showcode(void)
{
cout<<”object number”<<code<<”\n”;
}
static void showcount(void)     //static member function
{
cout<<”count”<<count<<”\n”;
}
};
int test::count;
int main()
{
test t1,t2;
t1.setcode();
t2.setcode();
test::showcount();
test t3;
t3.setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
return 0;
}
```

Output:

```
count:2
count:2
object number:1
object number:2
object number:3
```

CONTAINER CLASSES AND ITERATORS :

A container class is a data type that is capable of holding a collection of objects.

In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items.

Container classes commonly provide services such as insertion, deletion, searching, sorting and testing an item to determine whether it is a member of the collection.

Container classes typically implement a fairly standardized minimal set of functionality. Most well-defined containers will include functions that:

- Create an empty container (via a constructor)

- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Some example container classes are:

Arrays, stacks, queues, trees and linked list.

It is common to associate iterator objects-or more simply iterators-with container classes.

In C++, containers typically only hold one type of data. For example, if you have an array of integers, it will only hold integers. Unlike some other languages, C++ generally does not allow you to mix types inside a container. If you want one container class that holds integers and another that holds doubles, you will have to write two separate containers to do this.

USE OF CONTAINER:

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators

Types of container :

- Sequence container-Vector,dequeue,list
- Associate container- set,multiset,map,multimap
- Derived Container- stack,queue,priority queue

Container	**Description**	**Header File**	**Iterator**
vector	A dynamic array.	<vector>	Random Access

list	A linear list. Allows insertion and deletion anywhere.	<list>	Bidirectional
deque	A double-ended queue. Allows insertion and deletion at both ends.	<deque>	Random Access
set	A set in which each element is unique.	<set>	Bidirectional
multiset	A set in which each element is not necessarily unique.	<set>	Bidirectional
map	Stores key/value pairs in which each key is associated with only one value.	<map>	Bidirectional
multimap	Stores key/value pairs in which one key may be associated with two or more values.	<map>	Bidirectional
stack	A stack. LIFO	<stack>	No Iterator
queue	A queue. FIFO	<queue>	No Iterator
priority_queue	A priority queue. The first element out is always the highest priority element.	<queue>	No Iterator

SEQUENCE CONTAINER :

Sequence containers store elements in a linear sequence, like line. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operations on them.

Three types of Sequence container**s are**

Vector,dequeue,list

ASSOCIATE CONTAINERS :

Associate containers are designed to support direct access to elements using keys. They are not sequential.

4 types of Associate containers are ,

set,multiset,map,multimap

All these containers are store data in a structure called tree. Containers set and multiset can store a number of items and provide operations for manipulating them using the values as keys.Containers map and multimap are used to store pairs of items, one called the key and other called the value.

We can manipulate the values using the keys associated with them. The values are sometimes called mapped values.

DERIVED CONTAINERS :

Stack, queue, and priority queue are called derived containers. These are also called container adaptors.They support two member functions pop() and push() for implementing deleting and inserting operations.

APPLICATIONS OF CONTAINERS :

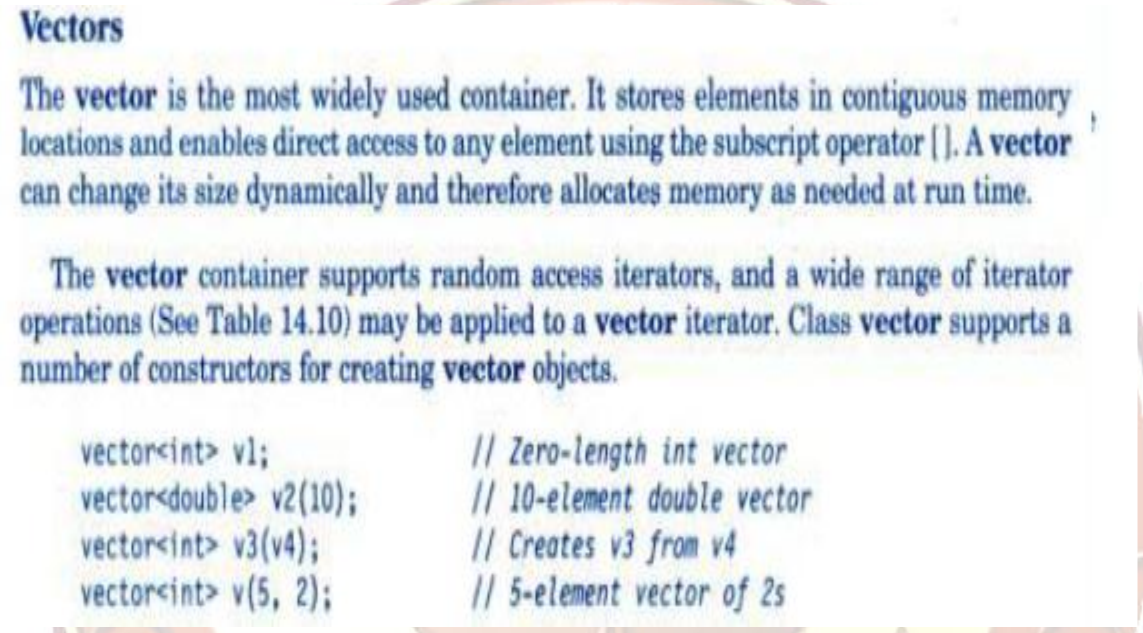
There are three main applications of containers mostly used are namely vector,list and map.

VECTOR:

It stores elements in continuous memory locations and enables direct access to any element using t subscript operator[]. A vector can change its size dynamically.

Vector<int>v1 ;   //zero length vector

Vector<double> v2(10) //10 element double vector

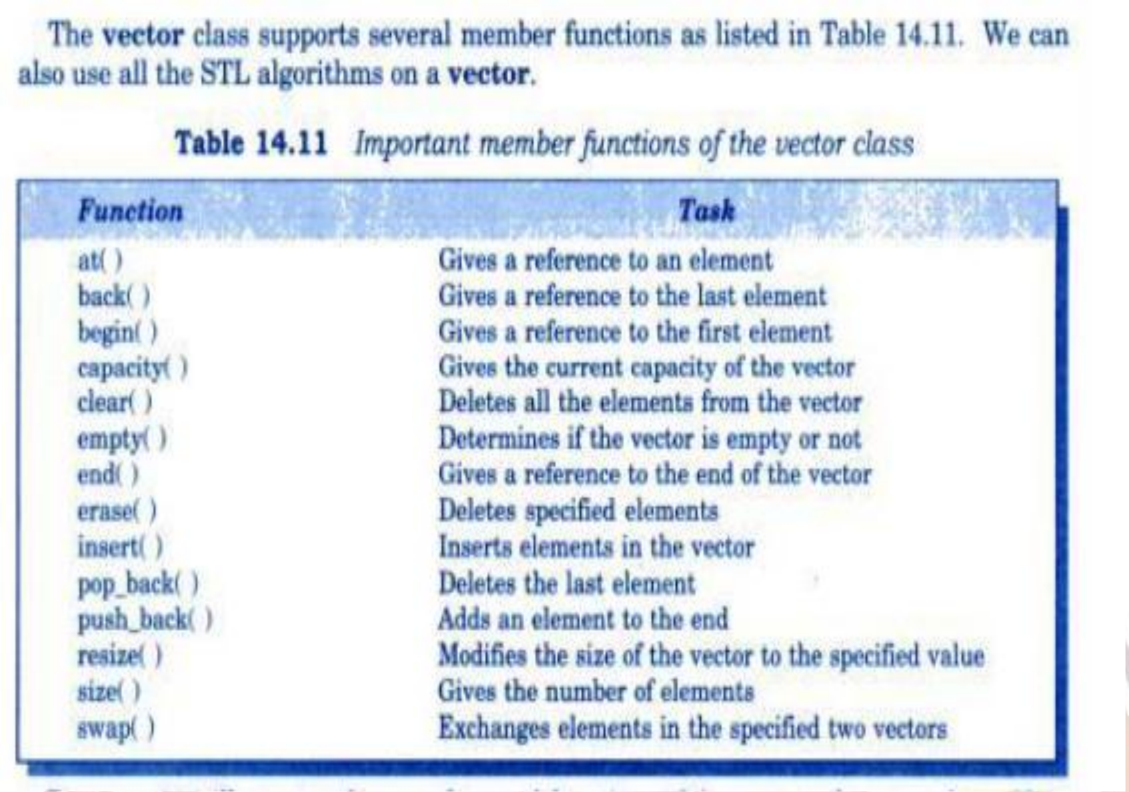


Vectors

The **vector** is the most widely used container. It stores elements in contiguous memory locations and enables direct access to any element using the subscript operator [ ]. A **vector** can change its size dynamically and therefore allocates memory as needed at run time.

The **vector** container supports random access iterators, and a wide range of iterator operations (See Table 14.10) may be applied to a **vector** iterator. Class **vector** supports a number of constructors for creating **vector** objects.

```
vector<int> v1;             // Zero-length int vector
vector<double> v2(10);      // 10-element double vector
vector<int> v3(v4);         // Creates v3 from v4
vector<int> v(5, 2);        // 5-element vector of 2s
```



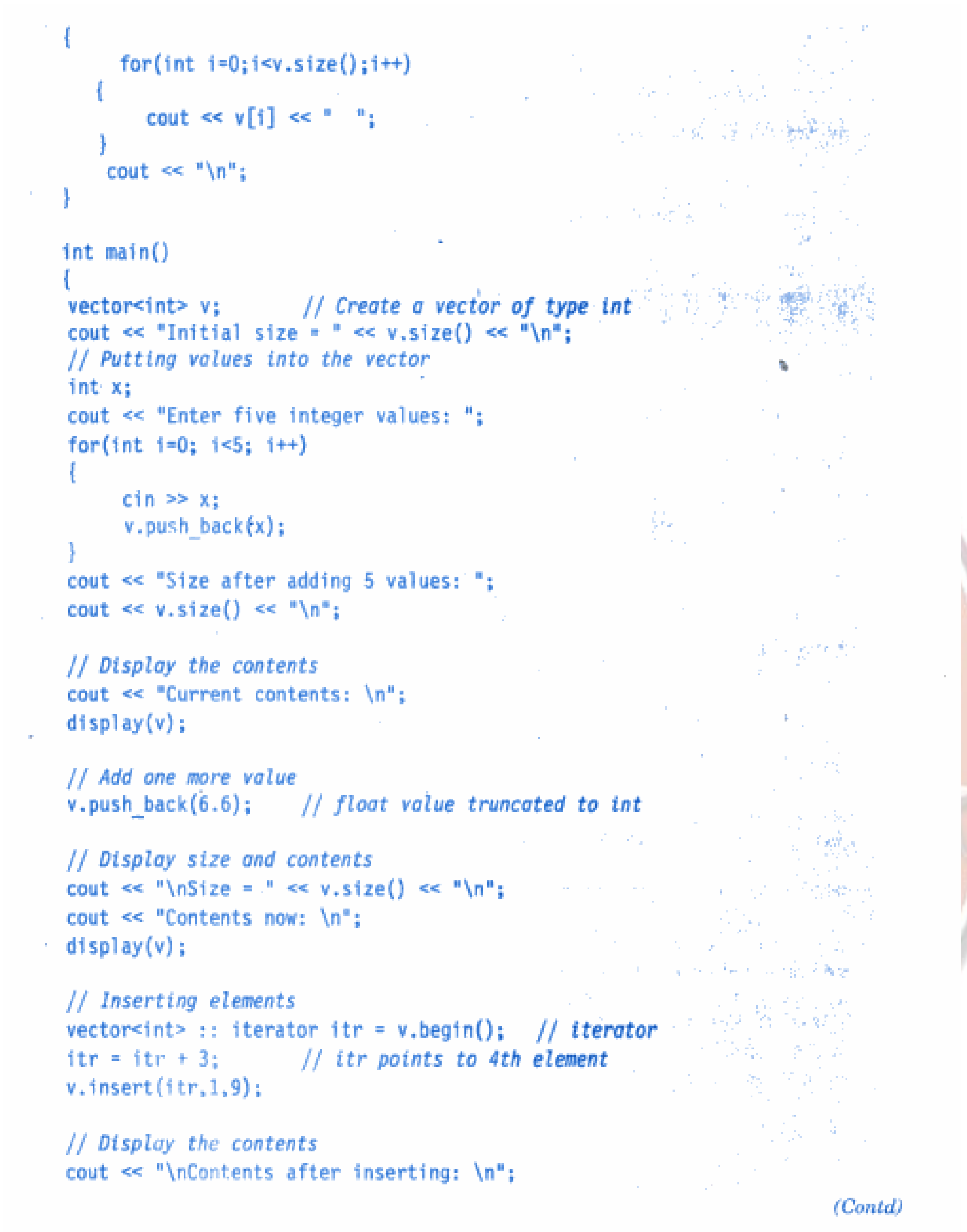
The **vector** class supports several member functions as listed in Table 14.11. We can also use all the STL algorithms on a **vector**.

Table 14.11 *Important member functions of the vector class*

Function	*Task*
at( )	Gives a reference to an element
back( )	Gives a reference to the last element
begin( )	Gives a reference to the first element
capacity( )	Gives the current capacity of the vector
clear( )	Deletes all the elements from the vector
empty( )	Determines if the vector is empty or not
end( )	Gives a reference to the end of the vector
erase( )	Deletes specified elements
insert( )	Inserts elements in the vector
pop_back( )	Deletes the last element
push_back( )	Adds an element to the end
resize( )	Modifies the size of the vector to the specified value
size( )	Gives the number of elements
swap( )	Exchanges elements in the specified two vectors

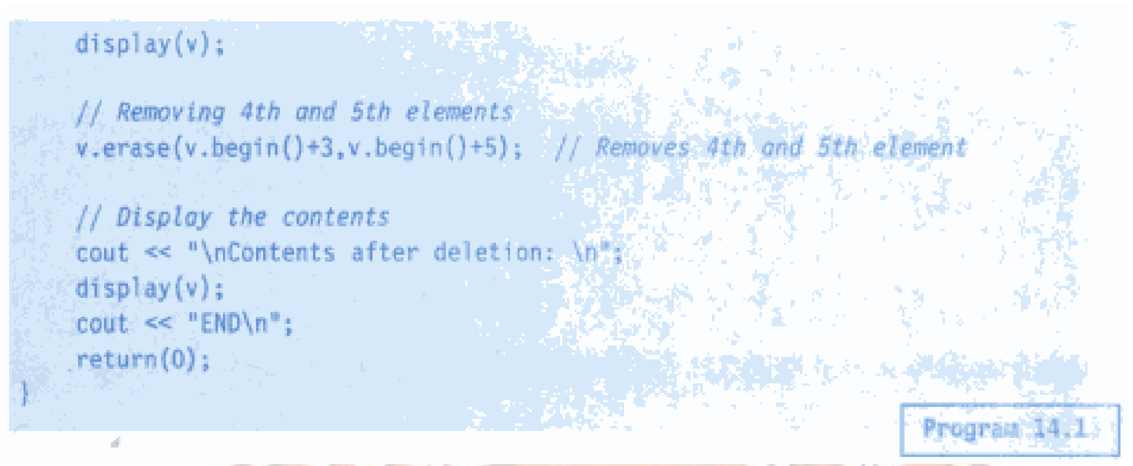
PROGRAM USING VECTOR

```
#include<iostream.h>
#include<vector>   // Vector header file
void display(vector<int> &v)
{
for (int i=0;i<v;i++)
{
   cout<<v[i] ;
```



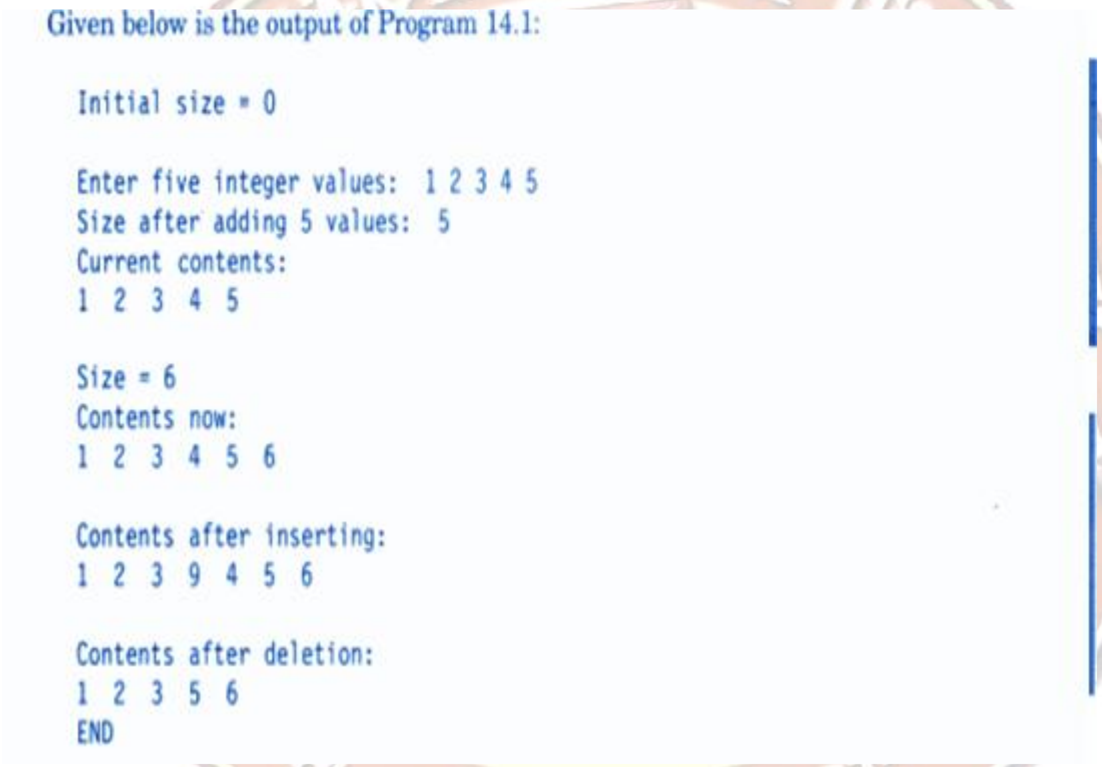
```
{
      for(int i=0;i<v.size();i++)
   {
         cout << v[i] << "  ";
   }
     cout << "\n";
}

int main()
{
vector<int> v;         // Create a vector of type int
cout << "Initial size = " << v.size() << "\n";
// Putting values into the vector
int x;
cout << "Enter five integer values: ";
for(int i=0; i<5; i++)
{
      cin >> x;
      v.push_back(x);
}
cout << "Size after adding 5 values: ";
cout << v.size() << "\n";

// Display the contents
cout << "Current contents: \n";
display(v);

// Add one more value
v.push_back(6.6);      // float value truncated to int

// Display size and contents
cout << "\nSize = " << v.size() << "\n";
cout << "Contents now: \n";
display(v);

// Inserting elements
vector<int> :: iterator itr = v.begin();   // iterator
itr = itr + 3;         // itr points to 4th element
v.insert(itr,1,9);

// Display the contents
cout << "\nContents after inserting: \n";
```

(Contd)



```
    display(v);

    // Removing 4th and 5th elements
    v.erase(v.begin()+3,v.begin()+5);  // Removes 4th and 5th element

    // Display the contents
    cout << "\nContents after deletion: \n";
    display(v);
    cout << "END\n";
    return(0);
}
```

Program 14.1



Given below is the output of Program 14.1:

```
Initial size = 0

Enter five integer values:  1 2 3 4 5
Size after adding 5 values:  5
Current contents:
1  2  3  4  5

Size = 6
Contents now:
1  2  3  4  5  6

Contents after inserting:
1  2  3  9  4  5  6

Contents after deletion:
1  2  3  5  6
END
```

LISTS:

It supports a bidirectional, linear list and provides an efficient implementation for deletion and insertion operations. Unlike a vector which supports random access, a list can be accesses sequentially only.

Class list provides many member functions for manipulating the elements of a list. Important member functions of the list class are given in the table below.

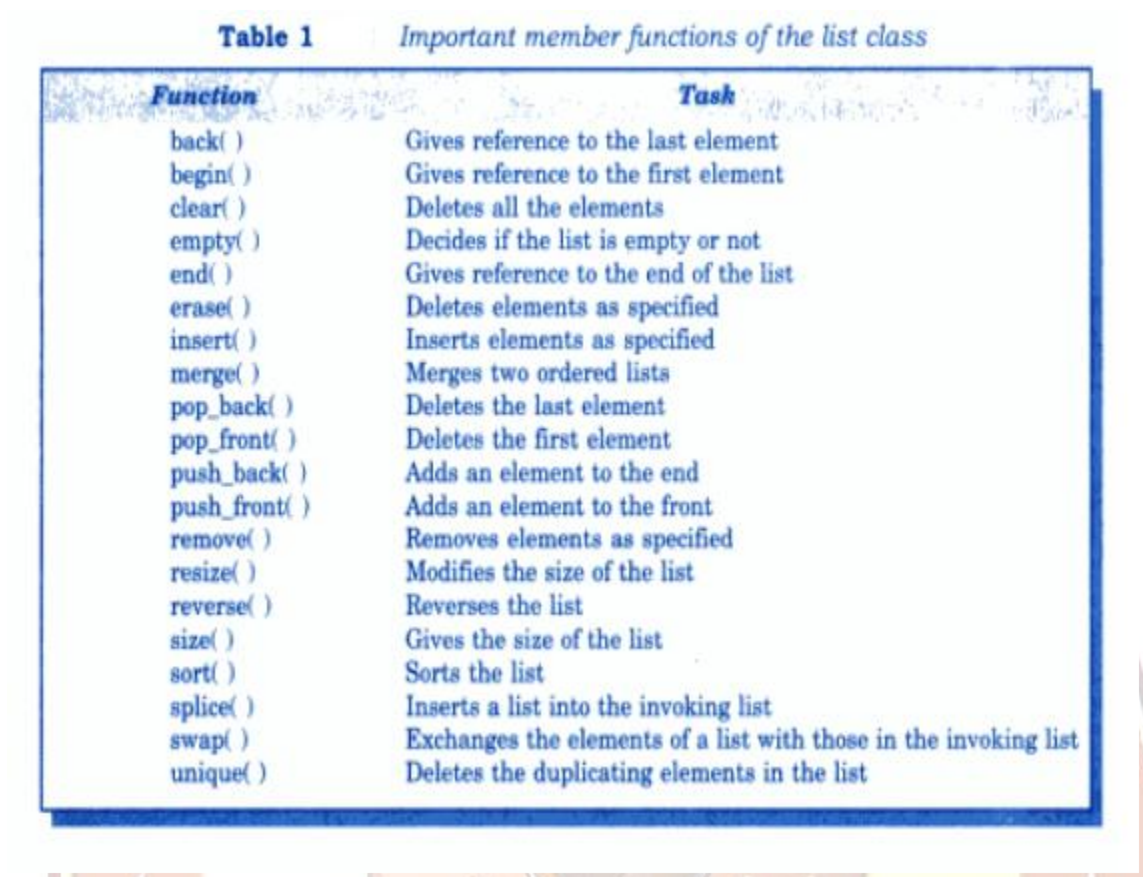


Table 1 *Important member functions of the list class*

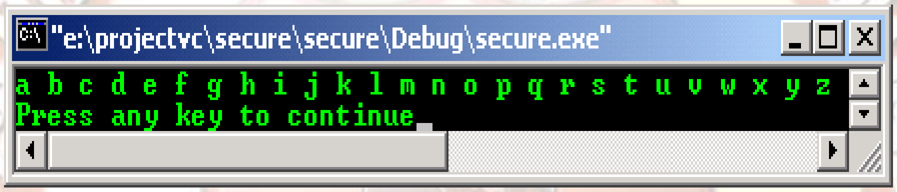
Function	***Task***
back( )	Gives reference to the last element
begin( )	Gives reference to the first element
clear( )	Deletes all the elements
empty( )	Decides if the list is empty or not
end( )	Gives reference to the end of the list
erase( )	Deletes elements as specified
insert( )	Inserts elements as specified
merge( )	Merges two ordered lists
pop_back( )	Deletes the last element
pop_front( )	Deletes the first element
push_back( )	Adds an element to the end
push_front( )	Adds an element to the front
remove( )	Removes elements as specified
resize( )	Modifies the size of the list
reverse( )	Reverses the list
size( )	Gives the size of the list
sort( )	Sorts the list
splice( )	Inserts a list into the invoking list
swap( )	Exchanges the elements of a list with those in the invoking list
unique( )	Deletes the duplicating elements in the list

- The following general list example creates an empty list of characters, inserts all characters from 'a' to 'z', and prints all elements by using a loop that actually prints and removes the first element of the collection:

```
// list example
#include <iostream.h>
#include <list>
int main()
{
   // list container for character elements
   list<char> elem;
   // append elements from 'a' to 'z'
   for(char ch='a'; ch<= 'z'; ++ch)
      elem.push_back(ch);
```

```
    // while there are elements, print and remove the element
    while(!elem.empty())
    {
       cout<<elem.front()<<' ';
       elem.pop_front();
    }
    cout<<endl;
    return 0;
}
```

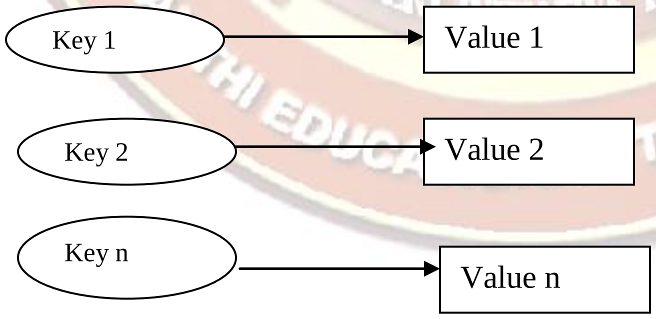
Output:

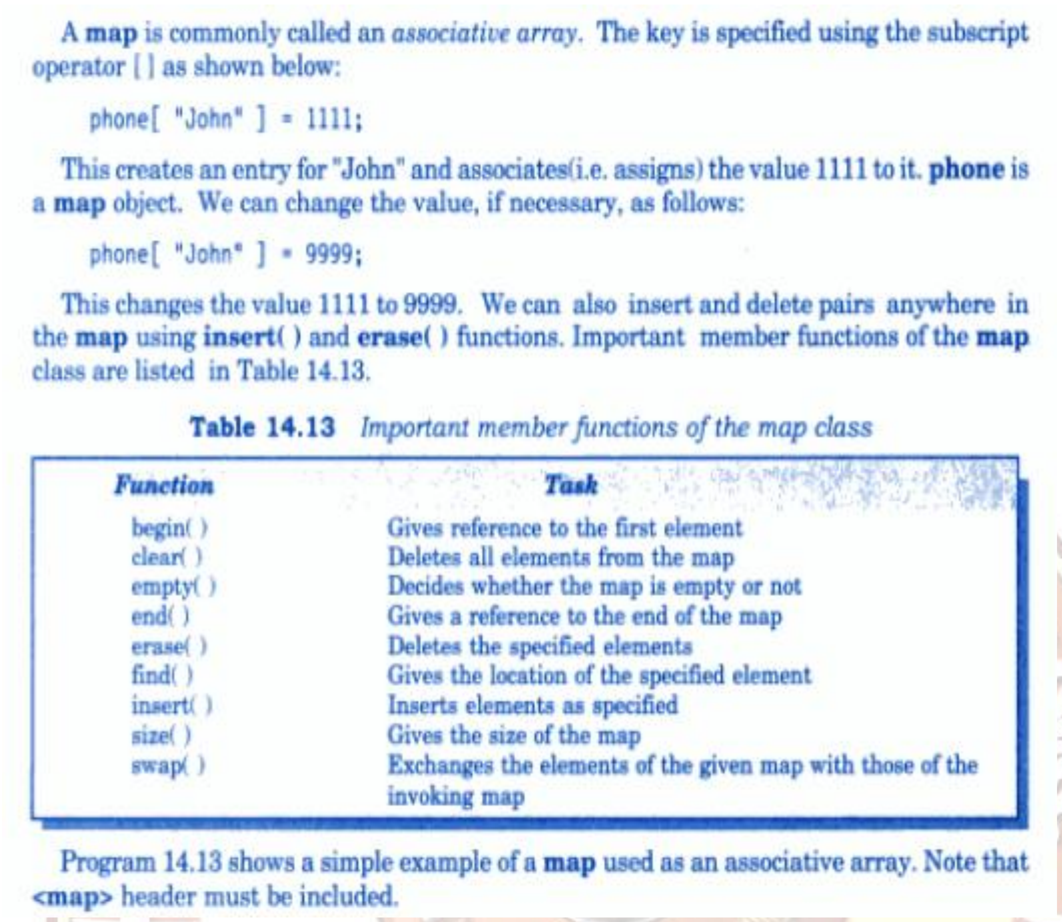


```
"e:\projectvc\secure\secure\Debug\secure.exe"
a b c d e f g h i j k l m n o p q r s t u v w x y z
Press any key to continue
```

MAP:

A map is a sequence of pairs where a single value is associated with each unique key.Retrival of value is based on the key and is very fast. A map is commonly called is associative array.





A **map** is commonly called an *associative array*. The key is specified using the subscript operator [ ] as shown below:

```
phone[ "John" ] = 1111;
```

This creates an entry for "John" and associates(i.e. assigns) the value 1111 to it. **phone** is a **map** object. We can change the value, if necessary, as follows:

```
phone[ "John" ] = 9999;
```

This changes the value 1111 to 9999. We can also insert and delete pairs anywhere in the **map** using **insert( )** and **erase( )** functions. Important member functions of the **map** class are listed in Table 14.13.

Table 14.13 *Important member functions of the map class*

Function	*Task*
begin( )	Gives reference to the first element
clear( )	Deletes all elements from the map
empty( )	Decides whether the map is empty or not
end( )	Gives a reference to the end of the map
erase( )	Deletes the specified elements
find( )	Gives the location of the specified element
insert( )	Inserts elements as specified
size( )	Gives the size of the map
swap( )	Exchanges the elements of the given map with those of the invoking map

Program 14.13 shows a simple example of a **map** used as an associative array. Note that **<map>** header must be included.

ITERATORS:

An iterator is an object that "walks through" a collection, returning the next item. Once an iterator for a class has been written, obtaining the next element from the class can be expressed simply.

Just as a book being shared by several people could have several bookmarks in i tat once,a container class can have several iterators operating on it at once.Each iterator maintains its own position information.

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

There are 5 types of iterator :

- **Input** –Used only to traverse in a container
- **Output-** Used only to traverse in a container

- **Forward**-Support all operations of input and output and also retains its position in the container.
- **Bidirectional**-Supporting all forward iterator operations, and provides ability to move in the backwared directions.
- **Random**-combines the functionality of Bidirectional with an ability to jump to an arbitrary location.

Different types of iterator must be used with the different types of containers.

PROXY CLASSES :

A proxy class is a class that acts on behalf of another class. There are many uses for proxy classes, but generally they are used to simplify a complex object's interface, by hiding details that are of no relevance within the object's current context.

For example we had classes: A that had method do(); and class B had method data();. If there is a way (using Boost Preprocessor for example) to create a proxy class that would have all methods from A and B (here do() data()) and a constructor taking in that pointers to that classes instances - one for A and one for B?

```
JOIN(A, B, C);// or if needed JOIN("path_to_A.h", "path_to_B.h", C)
//...
A * a = new A();
B * b = new B();
C * c = new C(a, b);
c->data();
c->do();
```

A proxy class that allows you to hide even the private data of a class from clients of the class. Providing clients of your class with a proxy class that knows only the public interface to your class enables the clients to use your class’s services without giving the clients access to your class’s implementation details.

<u>Implementing a proxy class requires several steps.</u>

First, we create the class definition for the class that contains the proprietary implementation we would like to hide.

<u>Example : XYZ class definition</u>

```
class XYZ
 {
 public:
 // constructor
XYZ( int v )
 :value( v ) // initialize value with v
 {
 // empty body
 } // end constructor XYZ
 // set value to v
void setvalue(int v)
 {
 value = v; // should validate v
 } // end function setValue
 // return value
int getvalue()const
 {
 return value;
 } // end function getValue
 private:
```

```
{
int value ;
 }; // end class XYZ
```

We define a proxy class called Interface with an identical public interface (except for the constructor and destructor names) to that of class Implementation. The proxy class's only private member is a pointer to an Implementation object. Using a pointer in this manner allows us to hide class XYZ'S implementation details from the client.

Example : Proxy class Interface definition.

```
Interface.h
 //Proxy class Interface definition.
//Client sees this source code, but the source code does not reveal the data layout of class //Implementation.
class Interface
{
public:
Interface( int ); // constructor
~Interface(); // destructor
private:
{
}; // end class Interface
```

The member-function implementation file for proxy class Interface is the only file that includes the header file implementation.h containing class implementation. It provided to the client as a precompiled object code file along with the header file Interface.h that includes the function prototypes of the services provided by the proxy class. Because file Interface.cpp is made available to the client only as object code, the client is not able to see the interactions between the proxy class and the proprietary class.

<u>Example: Interface class member-function definitions</u>

Interface.cpp

```
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.

#include "Interface.h" // Interface class definition
// constructor

class Implementation; // forward class declaration

Interface::Interface( int v )
void setValue( int ); // same public interface as
: ptr ( ) // initialize ptr to point to
int getValue() const;
{ // a new Implementation object
// empty body
} // end Interface constructor
// call Implementation's setValue function
void Interface::setValue( int v )
{
}
```

Define function overloading with a simple example. ***(NOV/DEC 2010) (MAY/JUNE 2013)***

17. What is operator overloading? Overload the numerical operators '+' and '/' for Complex numbers

"addition" and "division" respectively. ***(APRIL/ MAY 2011)***

OVERLOADING: FUNCTION OVERLOADING AND OPERATOR OVERLOADING:

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

FUNCTION OVERLOADING:

- Generally overloading means giving additional meaning to an existing operator or to an existing function.
- In function overloading a function with the same name is made to perform different functions at different situations.
- This is basically polymorphism and function overloading falls under the category of compile time polymorphism.

Example:

```
#include <iostream.h>
#include <conio.h>
void add(int,int);
void add(float,float);
void main()
```

```
{
clrscr();
int a=10,b=20;
float c=1.23,d=1.23;
add(a,b);
add(c,d);
getch();
}
void add(int a,int b)
{
int z;
z=a+b;
cout <<"\n The result of integer addition is\n"<<z;
}
void add(float a, float b)
{
float z;
z=a+b;
cout <<"\n The result of float addition is\n"<<z;
}
```

Output:

```
The result of integer addition is
30
The result of float addition is
2.46
```

- In the above program the function ‘add’ exhibited the polymorphism characteristic namely ‘function overloading’

- In this 2 function definition relating to the function ‘add’ was written,
  One for integer addition and another for float addition.
- When the function call is made the appropriate function definitions are linked
- This happens during compile time and so function overloading is a classification of compile time polymorphism.

OPERATOR OVERLOADING:

- Operator overloading means giving additional meaning to existing operators
- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.
- It doesn’t change the meaning and precedence of the original operator.
- Overloaded operator is used to perform operation on user-defined data type.
- Almost all the operators in c++ can be overloaded except the following.

Operators which cannot be overloaded

- sizeof ()
- Conditional operator (?:)
- Scope resolution operator (**::**)
- Class member access operator (.,.*)

SYNTAX:

```
return-type operator  op-symbol(argument list)
{
body of the function;
}
```

Generally there are **three general classifications** of operator namely

Unary Operator

binaryOperator

ternary Operator

Among the above unary and binary operators can be overloaded and ternary operator cannot be overloaded.

OVERLOADING UNARY OPERATOR:

Unary operators like unary + , Unary - ….. can be overloaded as follows

EXAMPLE:

```
#include <iostream.h>
#include <conio.h>
class unary
{
int a;
public:
void get()
{
a=10;
}
void show()
{
cout <<"\n The value of the object is\n"<<a;
}
void operator - ()
{
```

```
a= - a;
}
};
void main()
{
clrscr();
unary u;
u.get();
u.show();
-u;                    //unary operator - called
u.show();
getch();
}
```

OUTPUT:

```
The value of the object is
10
The value of the object is
-10
```

- In the above program, the unary operator minus is overloaded.
- In this an object ‘u’ is created and given the value 10
- When the unary operator – is called the operator function is invoked and the value of the member data of the object ‘u’ is negated.
- The negated value is then displayed.

OVERLOADING BINARY OPERATOR

- Binary operators manipulate with 2 operands
- It is overloaded in the following way.

EXAMPLE:

```
#include <iostream.h>
#include <conio.h>
class binary
{
        int a;
        int b;
public:
void get()
{
        a=10;
        b=20;
}
void show()
{
cout <<"\n The values of the object is\n"<<a<<"  "<<b;
}
binary operator + (binary x)
{
binary t;
t.a=a+x.a;
t.b=b+x.b;
return (t);
}
};
void main()
{
```

```
clrscr();
binary u,v,z;
u.get();
v.get();
z=u+v;
z.show();
getch();
}
```

OUTPUT:

```
The values of the objects are
    20    40
```

- In the above program the binary operator + is overloaded.
- In the main (), 2 objects namely 'u' and 'v' are added.
- The object 'u' calls the operator + and passes the object 'v' as argument.
- In the operator function the member data of the object 'u' and the object 'v' is added and the final result is stored in the member data of a temporary object 't' and passed back to the main().This is the reason why the return type of the operator function is binary.

PART A Question and Answers

1. Distinguish between Procedure Oriented Programming and Object Oriented Programming

Procedure Oriented programming	Object Oriented Programming
Emphasis is on algorithm.	Emphasis is on data rather than procedure Large
Programs are divided into smaller programs called functions.	Programs are divided into objects.
Functions share global data.	Functions that operate on the data of an object are tied together.
Data move openly around the system from function to function.	Data is hidden and cannot be accessed by external functions.
Employs top-down approach in program design.	Follows bottom-up approach

2. Define Object Oriented Programming (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.

3. Write any four features of OOPS.

•Emphasis is on data rather than on procedure.

•Programs are divided into objects.

•Data is hidden and cannot be accessed by external functions.

•Follows bottom-up approach in program design.

4. What are the basic concepts of OOS? (DEC 2007)

•Objects.

•Classes.

• Data abstraction and Encapsulation.

•Inheritance.

•Polymorphism.

•Dynamic binding.

•Message passing.

5. What are objects? How are they created? (DEC 2005) (NOV/DEC 2012)

Objects are basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Each object has the data and code to manipulate the data and theses objects interact with each other.

Objects are created by using the syntax:

```
classname obj1,obj2,…,objn;
(or) during definition of the class:
class classname
{
  -------
  -------
}obj1,obj2,…,objn;
```

6. What is a class? (DEC 2005)(NOV/DEC 2012)

The entire set of data and code of an object can be made a user-defined data type with the help of a class. Once a class has been defined, we can create any number of objects belonging to the classes. Classes are user-defined data types and behave like built-in types of the programming language.

7. Define Encapsulation and Data Hiding. (DEC 2005)(NOV/DEC 2011)(NOV/DEC 2013)

The wrapping up of data and functions into a single unit is known as data encapsulation. Here the data is not accessible to the outside world. The insulation of data from direct access by the program is called data hiding or information hiding.

8. What are data members and member functions?

Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, and cost and uses functions to operate on these attributes. The attributes are sometimes called as data members because they hold information. The functions that operate on these data are called as methods or member functions. Eg: int a,b; // a,b are data members Void getdata ( ) ; // member function

9. Give any four advantages/benefits of OOPS. (MAY 2007)

•The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

•It is possible to have multiple instances of an object to co-exist without any interference.

•Object oriented programming can be easily upgraded from small to large systems.

•Software complexity can be easily managed.

10. What are the features required for object-based programming Language?

•Data encapsulation.

•Data hiding and access mechanisms.

•Automatic initialization and clear up of objects.

•Operator overloading.

11. Give any four applications of OOPS (MAY 2007)

•Real-time systems.

•Simulation and modeling.

•Object-oriented databases.

•AI and expert systems.

12. Give any four applications of c++?

•Since c++ allows us to create hierarchy-related objects, we can build special object-oriented
libraries, which can be used later by many programmers.

•C++ easily maintainable and expandable.

•C part of C++ gives the language the ability to get close to the machine-level details.

•It is expected that C++ will replace C as a general-purpose language in the near future.

13. What are tokens?

The smallest individual units in a program are known as tokens. C++ has the following tokens, Keyword, Identifiers, Constants, Strings, Operator.

14. What are keywords?

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names from the program variables or other user defined program elements.

Eg: goto, If, struct, else, union etc.

15. Rules for naming the identifiers in C++.

•Only alphabetic characters, digits and underscore are permitted.

•The name cannot start with a digit.

•The upper case and lower case letters are distinct.

•A declared keyword cannot be used as a variable name.

16. What are the operators available in C++?

All operators in C are also used in C++. In addition to insertion operator << and extraction operator >> the other new operators inC++ are,

- :: 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

17. What is a scope resolution operator? What is the use of scope resolution operator (DEC 2007)

(APRIL/MAY 2010)

Scope resolution operator is used to uncover the hidden variables. It also allows access to global version of variables.

Eg:

```
#include<iostream.h>
int m=10; // global variable m
void main ( ){int m=20; // local variable m
cout<<”m=”<<m<<”\n”;
cout<<”: : m=”<<: : m<<”\n”;}
```

<u>output:</u>

```
20
10
```

(: : m access global m)Scope resolution operator is used to define the function outside the class.

Syntax:Return type <class name> : : <function name>Eg:Void x : : getdata()

18. What is meant by an expression? (APRIL/MAY 2008)

An expression is a combination of constant, variable, operators and function calls written in any form as per the syntax of the c++ language.

19. Define abstraction. (DEC 2005)

Creation of well-defined interface for an object, separate from its implementation. E.g., key functionalities (init, add, delete, count, print) which can be called independently of knowing how an object is implemented.

20. What is member-dereferencing operator?

C++ permits to access the class members through pointers. It provides three pointer-to-member operators for this purpose,

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

21. What is function prototype?

The function prototype describes function interface to the compiler by giving details such as number, type of arguments and type of return values Function prototype is a declaration statement in the calling program and is of the following

Type function_name(argument list);

Eg float volume(int x,float y);

22. What is an inline function ?(NOV/DEC 2011) (MAY/JUNE 2013)

An inline function is a function that is expanded in line when it is invoked. That is compiler replaces the function call with the corresponding function code. The inline functions are defined as

Inline function-header {function body}

23. Write some situations where inline expansion may not work

For functions returning values, if loop, a switch, or a goto exists for functions not returning values , if a return statement exists if function contain static variables, if inline functions are recursive.

24. What is a default argument ?

Default arguments assign a default value to the parameter, which does not have matching argument in the function call. Default values are specified when the function is declared.Eg : float amount(float principle, int period, float rate=0.15)Function call is Value=amount(5000,7);Here it takes principle=5000& period=7And default value for rate=0.15Value=amount(5000,7,0.34)Passes an explicit value 0f 0.34 to rate.We must add default value from right to left.

25. What are constant arguments ?

Keyword is const. The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointer . eg: int strlen (constchar *p);

26. How the class is specified?(OR) How a class declared in c++? (APRIL/MAY 2010)

Generally class specification has two parts class declaration It describes the type and scope of its member class function definition .It describes how the class functions are implemented.

The general form is Class

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

27. How to create an object?

Once the class has been declared, we can create variables of that type by using the class name Eg: class name x; //memory for x is created

28. How to access a class member?

Object-name is used to access a class member.

Function-name (actual arguments);

Eg: x.getdata (100,75.5);

29. How the member functions are defined?

Member functions can be defined in two ways outside the class definition Member function can be defined by using scope resolution operator ::

General format is

Return type class_Name :: function-name(argument declaration){}

Inside the class definition This method of defining member function is to replace the function declaration by the actual function definition inside the class. It is treated as inline function.

Eg:

```
class item
{
int a,b;
void getdata(int x,int y)
{
a=x;
b=y;
};
```

30. What is static data member?

Static variable are normally used to maintain values common to the entire class.Feature:It is initialized to zero when the first object is created. No other initialization is permitted only one copy of that member is created for the entire class and is shared by all the objects It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class .

It is stored separately rather than objects:

static int count//count is initialized to zero when an object is created.

int classname::count;//definition of static data member

31. What is static member function?

A member function that is declared as static has the following properties static function can have access to only other static member declared in the same class A static member function can be called using the class name as follows, **classname ::function_name;**

32. Define class and object (DEC 2005) (MAY/JUNE 2013)

Class: It is defined as blueprint or it is a collection of objects

Objects: is an instance of a class

Example:

```
class point
{
double x, y; // implicitly private
public:
void print();
void set( double u, double v );
};
```

33. When do we declare a member of a class static? (NOV/DEC 2009) (NOV/DEC 2013)

When the same data needs to be accessible by multiple instances of a class , the member of the class should be declared static.

34. What are the c++ operators that cannot be overloaded?(MAY 2007) (NOV/DEC 2012)

- → Size operator (sizeof)
- → Scope resolution operator (::)
- → Class member access operators(. , .*)
- → Conditional operator (?:)

35. What is called pass by reference?

In this method address of an object is passed, the called function works directly on the actual arguments.

36. Define constructor(NOV/DEC 2012)

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is same as class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class Eg: Class integer

```
{……public:
 integer( ); //constructor
………}
```

37. Define default constructor (NOV/DEC 2011)

The constructor with no arguments is called default constructor .

```
Eg:Class integer
        {
int m,n;
Public:Integer( );…….};
integer::integer( )//default constructor
{m=0;n=0;}
```

the statement integer a; invokes the default constructor.

38. Define parameterized constructor

Constructor with arguments is called parameterized constructor

```
Eg;Class integer
{ int m,n;
public:integer(int x,int y)
{m=x; n=y;
}
```

To invoke parameterized constructor we must pass the initial values as arguments to the constructor function when an object is declared.

This is done in two ways

1.By calling the constructor explicitly eg: integer int1=integer(10,10);

2.By calling the constructor implicitly eg: Integer int1(10,10);

39. Define default argument constructor

The constructor with default arguments are called default argument constructor

Eg:

Complex (float real, float imag=0);

The default value of the argument imag is 0.

The statement complex a(6.0) assign real=6.0 and imag=0

the statement complex a(2.3,9.0)assign real=2.3 and imag=9.0

40. What is the ambiguity between default constructor and default argument constructor?

The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it cause ambiguity for a statement such as A a; The ambiguity is whether to call A::A() or A::A(int i=0)

41. Define copy constructor

A copy constructor is used to declare and initialize an object from another object. It takes a reference to an object of the same class as an argument.

Eg: integer i2(i1); would define the object i2 at the same time initialize it to the values of i1. Another form of this statement is

Eg: integer i2=i1;

The process of initializing through a copy constructor is known as copy initialization.

42. Define dynamic constructor

Allocation of memory to objects at time of their construction is known as dynamic constructor. The memory is allocated with the help of the NEW operator

```
Eg: Class string
{
char *name; int length;
public:
string( )
{
length=0;
name=new char[length +1];
}
void main( )
{
string name1("Louis"),
name3(Lagrange);
}
```

43. Define destructor

It is used to destroy the objects that have been created by constructor. Destructor name is same as class name preceded by tilde symbol (~)

Eg;~integer(){}

A destructor never takes any arguments nor it does it return any value. The compiler upon exit from the program will invoke it. Whenever new operator is used to allocate memory in the constructor, we should use delete to free that memory.

44. Define multiple constructors (constructor overloading).

The class that has different types of constructor is called multiple constructors

Eg:

```
#include<iostream.h>
#include<conio.h>
class integer
{int m,n;
public:
integer( ) //default constructor
{m=0;n=0;}
integer(int a,int b) //parameterized constructor
{m=a; n=b;}
integer(&i) //copy constructor
{m=i.m;n=i.n;}
void main()
{
integer i1; //invokes default constructor
integeri2(45,67);//invokes parameterized constructor
integer i3(i2); //invokes copy constructor
}
```

45. Write some special characteristics / properties of constructor (NOV/DEC 2012)

•They should be declared in the public section

•They are invoked automatically when the objects are created

•They do not have return types, not even void and therefore, and they cannot return values

•They cannot be inherited, though a derived class can call the base class

•They can have default arguments

•Constructors cannot be virtual function

46. How the objects are initialized dynamically?

To call parameterized constructor we should the pass values to the object ie, for the constructor integer (int a, int b),it is invoked by integer a(10,18) this value can be get during run time. i.e., for above constructor

```
int p,q;
cin>>p>>q;
integer a(p,q);
```

47. Explain return by reference with an example.

A function can also return a reference. Consider the following function

```
int & max( int &x , int &y){ if(x>y)return x;else return y;}
```

Since the return type of max ( ) is int & the function returns reference to x or y (and not the values). Then a function call such as max (a , b) will yield a reference to either a or b depending on their values. The statement max (a , b) = -1; is legal and assigns –1 to a if it is larger, otherwise –1 to b.

48. What are Friend functions? Write the syntax(MAY/JUNE 2013)

A function that has access to the private member of the class but is not itself a member of the class is called friend functions. The general form is

friend data_type function_name( );

Friend function is preceded by the keyword 'friend'.

49. Write some properties of friend functions.

Friend function is not in the scope of the class to which it has been declared as friend. Hence it cannot be called using the object of that class. Usually it has object as arguments.

- It can be declared either in the public or private part of a class.
- It cannot access member names directly.

- It has to use an object name and dot membership operator with each member name. eg: (A.x)

50. What is function overloading? Give an example.

Function overloading means we can use the same function name to create functions that perform a variety of different tasks.

Eg: An overloaded add ( ) function handles different data types as shown below. // Declaration is

 int add(int a, int b); //add function with 2 arguments of same type int, int.
 int add(int a, int b, int c); //add function with 3 arguments of same type int, int, int.
double add(int p, double q); //add function with 2 arguments of different type
//Function calls
add (3 , 4); //uses prototype (i)
add (3, 4, 5); //uses prototype (ii)
add (3 , 10.0); //uses prototype (iii)

51. What is operator overloading?

C++ has the ability to provide the operators with a special meaning for a data type. This mechanism of giving such special meanings to an operator is known as Operator overloading. It provides a flexible option for the creation of new definitions for C++ operators.

The general form is

 return type classname :: operator (op-arglist){function body}

where

return type is the type of value returned by specified operation.
op - operator being overloaded. The op is preceded by a keyword operator.
operator op is the function name.

52. What are the advantages of operator overloading?(NOV/DEC 2010)

- Perform different operations on the same operands
- Makes code much more readable
- Extensibility: An operator will act differently depending on the operands provided.

53. Write at least four rules for Operator overloading.

Only the existing operators can be overloaded.

The overloaded operator must have at least one operand that is of user defined data type.

The basic meaning of the operator should not be changed.

Overloaded operators follow the syntax rules of the original operators.

They cannot be overridden.

54. How an overloaded operator can be invoked using Friend functions?

In case of unary operators, overloaded operator can be invoked as Operator op (x); In case of binary operators, overloaded operator can be invoked as Operator op (x , y)

55. List out the operators that cannot be overloaded using Friend function.

•Assignment operator =

•Function call operator ( )

•Subscripting operator [ ]

•Class member access operator →

56. What is meant by casting operator and write the general form of overloaded casting operator.

A casting operator is a function that satisfies the following conditions. It must be a class member. It must not specify a return type. It must not have any arguments.

The general form of overloaded casting operator is

```
operator type name ( )
{……….. // function statements
}
```

It is also known as conversion function.

57. What are the iteration or looping statements used in C++?

While: repeats a statement or block while its controlling expression is true.

Syntax: **while(condition){ //body of loop }**

do-while : Executes its body at least once

Syntax: **do{ //body of loop}while(condition);**

for : consists of three portions initialization, conditon, termination.

Syntax: for (initialization, condition, termination){ //body }

58. What is the difference between break & continue statements?

Break: We can force immediate termination of a loop, bypassing the conditional, the loop expression & any remaining code in the body of the loop. When a break statement is encountered in a loop, the loop is terminated & the program control resumes at the next statement following the loop. **Continue:** useful to force early termination. it continue running the loop, but stop processing the remainder of the code in its body for this particular iteration

59. What are the uses of break statements?

1. It terminates a statement sequence in switch statement.
2. It can be used to exit a loop.
3. It can be used as a civilized form of goto.

60. Mention some of the restrictions while using static keyword?

•They can call other static methods.

•They must only access the static data.

•They cannot refer to

– this
– or
– super
– anyway.

61. Define data members and member functions. (NOV/DEC 2011)

The attributes in the objects are known as data members because they hold the information. The functions that operate on these data are known as methods or member functions.

62. Write the properties of static member function? (NOV/DEC 2011)

- can be called, even when a class is not instantiated
- cannot be declared as virtual.
- cannot access THIS pointer.
- Can access only – static member data, static member functions, data and functions outside the class.

63. When do we declare a member of a class static? (Nov/Dec 2009)

When the same data needs to be accessible by multiple instances of a class , the member of the class should be declared static.

64. What are the advantages of operator overloading?(OR) Why is it necessary to overload an operator? (Nov/Dec 2009) (Nov/Dec 2010)

Giving additional meaning to existing operators is known as Operator overloading. By operator overloading an existing operator can be made to perform different operations than the stipulated one. It doesn’t change the meaning and precedence of the original operator.

Overloading is convenience for the programmer, allowing the development of operators for user-defined data types.

65. What is scope resolution operator and how can it be used for global variable? (April/May 2011) (April/May 2010)

In C, the global version of the variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator:: called the scope resolution operator. It is used to uncover a hidden variable.

Syntax: :: variable name;

66. What is Proxy class?

A proxy class is a class that acts on behalf of another class. There are many uses for proxy classes, but generally they are used to simplify a complex object's interface, by hiding details that are of no relevance within the object's current context. Although it may be better to declare a stripped-down version of the object, this is only practical when you have access to the riginal class implementation (in which case the full-blown class can be derived from the stripped-down class). Rather than re-invent wheels, accessing an existing class by proxy offers a more convenient workaround. Reference counting mechanisms are another example of proxy classes.

67. Write a C++ code to swap values of two variable using reference variables in function. (Nov/Dec 2014)

<u>Program:</u>

```
#include <iostream.h>

// function declaration
void swap(int &x, int &y);

int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;

/* calling a function to swap the values using variable reference.*/
swap(a, b);

cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;

return 0;
}
```

Output:

```
Before swap, value of a:        100
Before swap, value of b:        200
After swap, value of a:         200
After swap, value of b:         100
```

68. Write a C++ code to display "pen object instantiated" and "pen object destroyed" when class for pen constructor and destructor are called. (Nov/Dec 2014)

```
#include<iostream.h>
class pen
{
public:
  //constructor
  pen() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }
  //Destructor
  ~pen() {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};
int main( )
{
  pen p1;
  pen p2;
  return 0;
}
```

PART – B (University Questions)

1. State the merits and demerits of object oriented methodology. **(NOV / DEC 2007)**

2. Explain the following concepts of object oriented programming in detail with example. **(NOV / DEC 2007)**

(i). Data abstraction

(ii). Inheritance

(iii). Polymorphism

(iv). Object

3. State the rules to be followed while overloading an operator. Write a program to Illustrate overloading. **(NOV / DEC 2007)**

4. (i) Describe the application of OOPs Technology. **(APR / MAY 2007)**

(ii) What is an inline function? **(APR / MAY 2007)**

5. Illustrate the use of copy constructor and function overloading with C++ Program **(NOV/DEC 2011)**

6. Compare and contrast the following control structure with example: **(NOV / DEC 2007)**

(i). if -- else statement & switch statement.

(ii). Do – while and the while statement.

7. What is a friend function? What are the merits and demerits of using friend Function? **(NOV/DEC2009)**

8. Define a class ‘string’. Use overload ‘= =’ operator to compare two strings. **(NOV/DEC 2009)**

9. What is a conversion function? How is it create? Explain its syntax. **(NOV/DEC 2009)**

10. Give the syntax and usage of the reserved word inline with two examples. **(APRIL/MAY 2010)**

11. Explain the importance of constructors and destructors with example. **(APRIL/MAY 2010)**

12. Compare and contrast Structured Programming and Object Oriented Programming. **(NOV/DEC 2010)**

13. Distinguish between Data Encapsulation and Data Abstraction. **(NOV/DEC 2010)**

14. Mention the purpose of Constructor and Destructor functions. **(NOV/DEC 2010)**

15. Explain the control structures of C++ with suitable examples. **(NOV/DEC 2010) (MAY/JUNE 2013)**

16. Define function overloading with a simple example. **(NOV/DEC 2010) (MAY/JUNE 2013)**

17. What is operator overloading? Overload the numerical operators '+' and '/' for Complex numbers "addition" and "division" respectively. **(APRIL/ MAY 2011)**

18. Define friend class and specify its importance. Explain with suitable example. **(APRIL/MAY 2011)**

19. Explain the merits and demerits of object oriented paradigm. **(NOV/DEC 2011)**

20. Write a C++ program to define overloaded constructor and to perform string Initialization and string copy **(NOV/DEC 2011)**

21. Explain the concept of passing by reference using reference variables (6) **(APRIL/MAY 2011)**

22. Write a C++ program to assign 'n' projects to 'm' programmers based on the skill set of programmers using friend function. Use static variable to count total number of assignments (10)**(APRIL/MAY2011)**

23. Define 'Copy constructor' and 'Dynamic constructor'. What are the different ways of writing copy constructor? (6) **(APRIL/MAY 2011)**

24. Where to use friend function in binary operator overloading? How? Explain with an example. (10 **(APRIL/MAY 2011)(NOV/DEC 2012)**

25. Explain the concept of "passing array of objects as an argument" with an example

26. Write a program to evaluate the equation, A = B * C using classes and objects where A, B and C are objects of the same class (8)

27. Write a menu driven program to accept 2 integer and an operate (+,-,*, %,/) and to perform the operation and print the result.**(NOV/DEC 2012)**

28. Specify a class called complex to represent complex numbers. Overload +,-and * operators when working on the objects of this class**. (NOV/DEC 2012)**

29. Write short notes on the following

i. Comparison of conventional programming and oops

ii. Operator Overloading

iii. Constructor and Destructor **(MAY/JUNE 2013)**

30. Explain with examples the types of constructors in C++ **(NOV/DEC 2013)**

31. Write a C++ Program that contains a class String and overloads the following operators on Strings.

+ to concatenate two strings

-To delete a substring from the given string

== to check for the equivalence of both strings **(NOV/DEC 2013)**

32. Write a member function and friend function to subtract two complex numbers in C++.**(NOV/DEC 2014)**

33. Write a member function to perform matrix addition, simple addition and string concatenation by overloading +operator. **(NOV/DEC 2014)**

*********ALL THE BEST*********