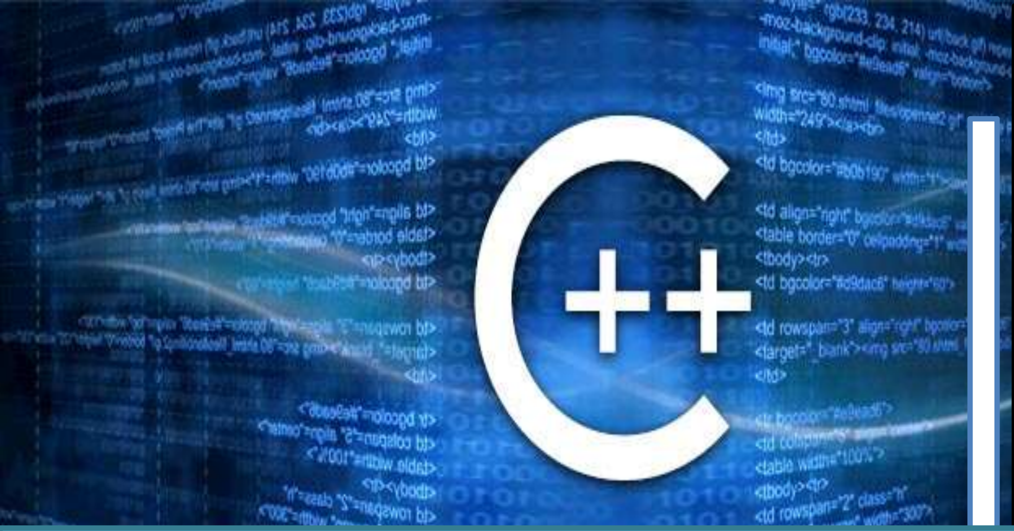
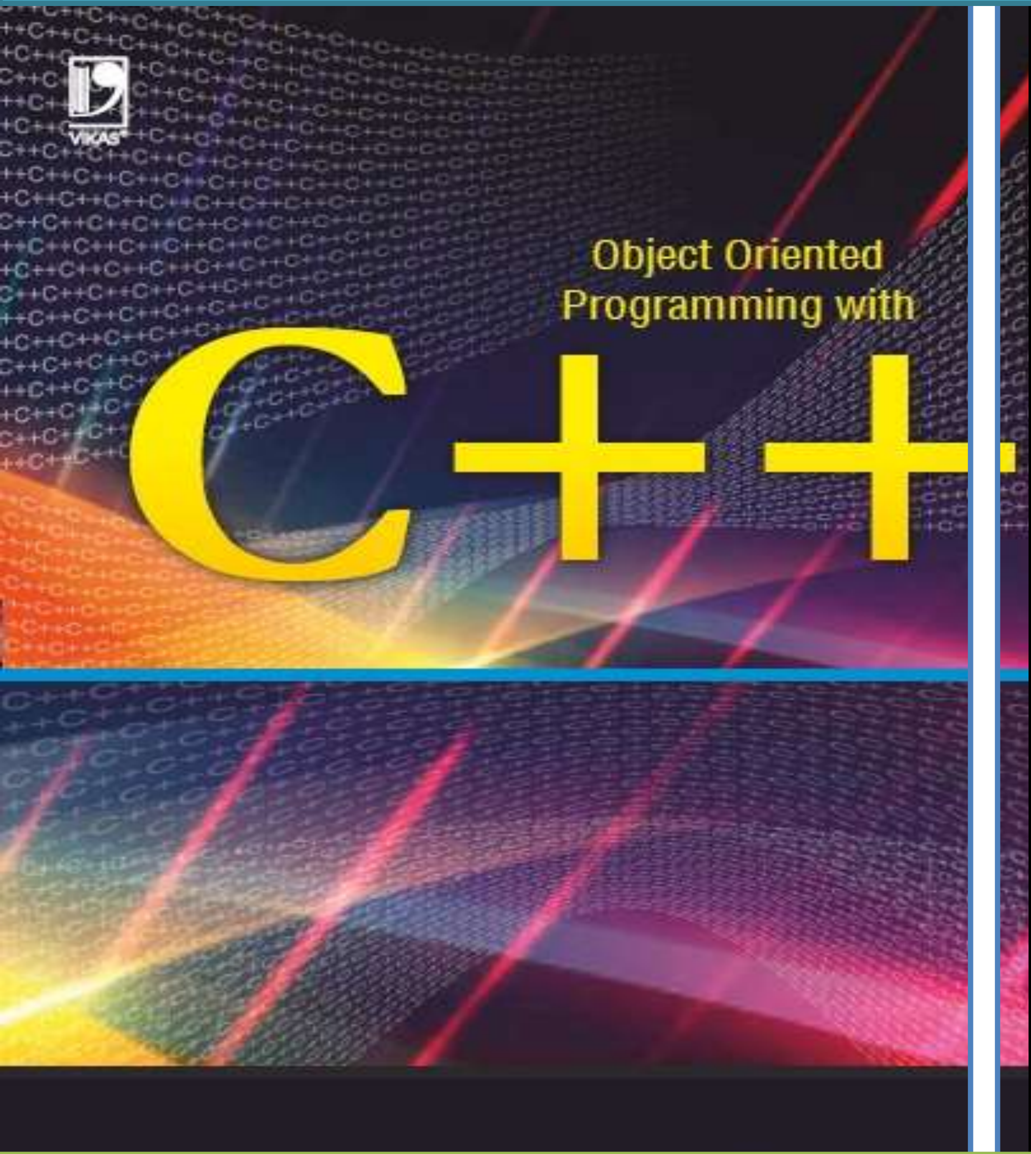


2016

A Guided Book Of Object Oriented Programming With CPP



ABU SALEH MUSA MIAH (ABID)

A Guidebook of

CPP

Engr. Abu Saleh Musa Miah (Abid)
Lecturer, Rajshahi Engineering Science and Technology College(RESTC)
B.Sc.Engg. in Computer Science and Engineering(**First Class First**)
M.Sc.Engg.(CSE-Pursuing),University of Rajshahi.
Email:abusalehcse.ru@gmail.com
Cell:+88-01734264899

Object Oriented Programming with CPP

Writer	:	**Engr. Abu Saleh Musa Miah (Abid).** B.Sc.Engg. in Computer Science and Engineering(**First Class First**) **M.Sc.Engg**.(CSE-Pursuing),University of Rajshahi. **Cell:+88-01734264899**
Email	:	abusalehcse.ru@gmail.com
Publisher	:	**Abu Syed Md Mominul Karim Masum.** **Chemical Engr.** Chief Executive Officer (CEO) Swarm Fashion, Bangladesh. Mohakhali,Dhaka.
Email	:	swarm.fashion@gmail.com
Cover page Design	:	Md. Kaiyum Nuri Chief Executive Officer (CEO) Jia Shah Rich Group(Textile) MBA,Uttara University
First Publication	:	Octobor-2016
Copyright	:	Writer
Computer Compose	:	Writer
Print	:	Royal Engineering Press & Publications. Meherchandi, Padma Residential Area, Boalia, Rajshahi.
Reviewer Team	:	Engr. Syed Mir Talha Zobaed. B.Sc. Engg. (First Class First) M.Sc. Engineering (CSE) University of Rajshahi Omar Faruk Khan (Sabbir) M. Engineering (CSE) University of Rajshahi
PRICE	:	400.00 TAKA (Fixed Price). US 05 Dollars (**Fixed Price**).

OOP CPP	:	Engr. Abu Saleh Musa Miah (Abid).
Published by	:	Royal Engineering Press, Bangladesh Meherchandi, Padma R/A, Boalia, Rajshahi.

Dedicated

To

All my Student's of Rajshahi Engineering Science and Technology college.
(Sharmin,Farjana,Pritul,Sony,Monika,Sarwar,Jery,Mitu,Joyonto,
Abdulla,Sobur,Jony,Eva,Shishir,Mishkat,Rafi,Tuhin)

[This Page is Left Blank Intentionally]

লেখকের কথা

Object Oriented Programming with CPP কোর্সটি বাংলাদেশের প্রায় সকল বিশ্ববিদ্যালয়ের সিলেবাসে স্থান লাভ করেছে,বাজারে টেক্সটবই থাকলেও গভীর ভাবে অনুধাবন ও পরীক্ষায় ভাল মার্কস উঠানোর মত সাজানো টেক্সটবই সহজলভ্য নয় । পরীক্ষার প্রস্তুতির স্বার্থে তাই, এই গাইডবইটি আপনাকে সহযোগিতা করবে বলে আশা করা যায় ।

এই বইয়ের পাঠক হিসেবে, আপনিই হচ্ছেন সবচেয়ে গুরুত্বপূর্ণ সমালোচক বা মন্তব্যকারী । আর আপনাদের মন্তব্য আমার কাছে মূল্যবান, কারন আপনিই বলতে পারবেন আপনার উপযোগী করে বইটি লেখা হলো কিনা অর্থাৎ বইটি কিভাবে প্রকাশিত হলে আরও ভাল হতো । সামগ্রিক ব্যাপারে আপনাদের যে কোন পরামর্শ আমাকে উৎসাহিত করবে ।

Engr. Abu Saleh Musa Miah (Abid)
Writer

ACKNOWLEDGEMENTS

I wish to express my profound gratitude to all those who helped in making this book a reality; especially to my families, the classmates, and authority of Royal Engineering Publications for their constant motivation and selfless support. Much needed moral support and encouragement was provided on numerous occasions by my families and relatives. I will always be grateful, to the numerous web resources, anonymous tutorials hand-outs and books I used for the resources and concepts, which helped me build the foundation.

I would also like to express my profound gratitude to Engr. Syed Mir Talha Zobaed for his outstanding contribution in inspiring, editing and proofreading of this guidebook. I am also grateful to Omar Faruque Khan (Sabbir) for his relentless support and guideline in making this book a reality.

I am thankful to the following readers those have invested their valuable times to read this book carefully and have given suggestion to improve this book:

Engr. Mainuddun Maruf (Principle,RESTC. B.Sc. Engg in Electrical and Electronics Engineering. IUT,)
Md.Moyeed Hossain (B.Sc.Engr. Computer Science and Engineering ,RUET, Lecturere, RESTC).
Md.Shamim Akhtar (B.A Honours,English,M.A. Lecturere, RESTC)
……………………………... And numerous anonymous readers.
I am also thankful to the different hand notes from where I have used lots of solutions, such as Dynamic Memory Allocation,Operator overloading etc

I wish to express my profound gratitude to the following writers whose books I have used in my Guidebooks:

1	Herbert Schildt	**C++: The Complete Reference Third Edition**
2	Steve Ouallin	Practical Programming
3	H. Schidt	**C++: The Complete Reference**, *McGraw Hill*
4	N. Barkakati	**Object Oriented Programming with C++**, *Prentice Hall India*
5	B. Stroustrap	**The C++ Pr**

…………………... and Numerous anonymous Power Point Slides and PDF chapters from different North American Universities.

Object Oriented Programming with CPP Syllabus

CSE1221: Object Oriented Programming with C++
75 Marks [70% Exam, 20% Quizzes/Class Tests, 10% Attendance]
3 Credits, 33 Contact hours, Exam. Time: 4 hours

Introduction: Object oriented programming and procedural oriented programming, encapsulation, inheritance, polymorphism, data abstraction, data binding, static and dynamic binding, message passing.

C++ as an object oriented language: Declaration and constants, expression and statements, data types, operator, Functions.

Classes: structure of classless. public, private and protected members, array of object, argumented member function, and non-augmented objects, nested member class and their object, pointer objects and pointer members, object a argument of function, static class member and static class. Friend function, friend class,

Inheritance: mode of inheritance, classifications of inheritance, virtual inheritance.

Array of objects of derived class.

Constructor and destructors: default constructor, argumented constructor, copy constructor, dynamic constructor, constructor function for derived class and their order of execution, destructor.

Operator and function overloading, unary and binary operator overloading, run-time and compile time polymorphism, object pointer and pointer to an object, virtual function, dynamic binding.

C++ data file: C++ file stream classes, input and output file, mode of files, file pointer, random file accessing,

Template and Exception handling: function template and class template, Exception Handling

Books Recommended:

1.	H. Schidt	:	**C++: A Beginner's Guide,** *McGraw Hill*
1.	H. Schidt	:	**C++: The Complete Reference**, *McGraw Hill*
2.	N. Barkakati	:	**Object Oriented Programming with C++,** *Prentice Hall India*
3.	B. Stroustrap	:	**The C++ Pr**

Index

Study Outlines for OOP with CPP

1. Differentiate between C and C++.When to use C++?
2. Write the general program structure of C++ and explain each section **Ex-2013**
3. Explain about different phases for executing a C++ program? **Ex-2013**
4. Write different data types used in C++.What is data abstraction? **Ex-2013**
5. Define Object Oriented programming (OOP). What are features of OOP? **Ex-2011**
6. What is ment by object oriented Programming(OOP)?Write the features of OOP. **Ex-2013**
7. what is object oriented programming? Why is it effective over structured programming? Explain. **Ex-2014**
8. Briefly discuss the properties of oop language **Ex-2014**
9. how can you allocated and free memory to an object? Explain with example? **Ex-2014**
10. How data hiding is accomplished in C++?Explain. **Ex-2013**
11. How are data and functions organized in OOP?Explain. **Ex-2013**
12. What are difference between OOP and Procedural Programming? **Ex-2011**
13. What is class ? What is an object? **Ex-2011-2013**
14. .write the general from of a class . differentiate between class and object. **Ex-2014**
15. what is a nested class? Why can it be useful?
16. What are the difference between 'class' and 'structure'? **Ex-2011**
17. What are 'class' and 'object'? **Ex-2011**
18. . what is friend function? Why ist it used ? describe with and example? **Ex-2014**
19. What is a friend? Do friends violate encapsulation? **Ex-2012**
20. What are some advantages / disadvantages of using friend functions? **Ex-2012**
21. what is in-line function ? what are the restrictions to in-line functions? **Ex-2014**
22. What is inline function? **Ex-2012**
23. How do you tell the compiler to make a member function inline? **Ex-2012**
24. How do you tell the compiler to make a member function inline? **Ex-2012**
25. What are advantages of using 'static member variable' and 'static member function' of a class? **Ex-2011**
26. Give an example of 'nested class member'. **Ex-2011**
27. Define constructor and destructor?
28. what are constructor and destructor function? Can these functions have input parameters and return type? **Ex-2014**
29. Explain the accessing mechanism of data members and member functions in case of Inside main() function and Inside a member function of the same class. **Ex-2013**
30. When do you declare a member of a class static

Find the error(s) in each of the following and explain how to correct it:

i. void ~Time(int);

ii. The following is a partial definition of class time:

```
class Time{
public:
// function prototypes
Private:
 int hour=0;
 int minute=0;
 int second=0;
```

iii. Assume the following prototypes is declared in class Employee:

int Employee (const char * , const char *); **Ex-2012**

31. Find the errors in the following class and explain how to correct them:

```
class Example{
public:
Example (int y=10)
  : data(y)
{
   //empty body
}
int get Incremented Data ( ) const
{
  Return ++ data;
}
Static int get Count( )
{
  cout<< "Data is"<<data<<end1 ;
  return count;
}
Private:
  int data;
  static int count ;
}; // end class Example
```

Ex-2012

32. What is the difference between the effects of the following two declarations:

```
Ratio y(x) ;
Ratio y=x ;
```
Ex-2012

33. What is the difference between the effects of the following two lines:

```
Ratio y=x ;
Ratio y;   y=x ;
```
Ex-2012

34. Given the following partial class , add the necessary constructor functions so that both 4 declarations within **main** ( ) are valid.

```
Class samp {
   int a;
public:
    //add constructor functions
     int get _a( ) {return a ; }
};
int main( ) {
      samp ob (88); // init ob's a to 88
```

samp ob array [10]; //noninitialized 10-element array//....}

35. To illustrate exactly when an object is constructed and destructed when returned from a function , create a class called **who** . Have **who's** constructor take one character argument that will be used to identify an object . Have the constructor display a message similar to this when constructing an object:

    constructing who# x

    where x is the identifying character associated with each object . When an object is destroyed , have a message similar to this displayed :

    Destroying who#x

    Where, again, x is the identify character . Finally , create a function called **make_who** ( ) that returns a who object . Give each object a unique name. Note that output displayed by the program. **Ex-2012**
36. How are memory allocated for different type of members of a class? **Ex-2011**
37. What is "this" pointer? **Ex-2012**
38. Can an array of objects be initialized?Explain. **Ex-2013**
39. What is "this"?Explain with an example. **Ex-2013**
40. What is new and delete?What are their advantages? **Ex-2013**
41. What is reference?Describe with example. **Ex-2013**
42. Can an array of objects be initialized?Explain.
43. What is "this"?Explain with an example. **Ex-2013**
44. What is new and delete?What are their advantages? **Ex-2013**
45. What is reference?Describe with example. **Ex-2013**
46. Write the output for the following lines

```
std::cout <<i << '\n' ;
std::cout << "i" << '\n' ;
std::cout << i /j << '\n' ;
std::cout <<"i=" << i ;
```

47. What is copy constructor? When copy constructor called?
48. What is default argument? **Ex-2011**
49. in c++ there are two ways to achieve call by reference . what are they? Discuss with example. **Ex-2014**
50. What is inline function? **Ex-2012**
51. why might the following function not be in-lined by your compiler?

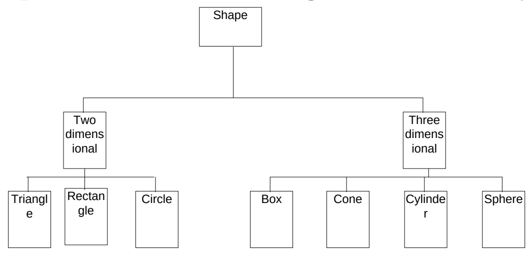
```
Void f1( )
                {
   int i ;
   for(i=0;i<10;i++)  cout<< i ;
}
```

    Ex-2012
52. What do you mean by operator overloading? What are the restrictions applied to operator overloading ? **Ex-2012**
53. What is operator overloading ? Why is it necessary? **Ex-2011**
54. Why overloading sometimes caused ambiguity? Which type of ambiguity may arise? **Ex-2011**
55. What are the advantages of overloading an operator using friend function? **Ex-2012**

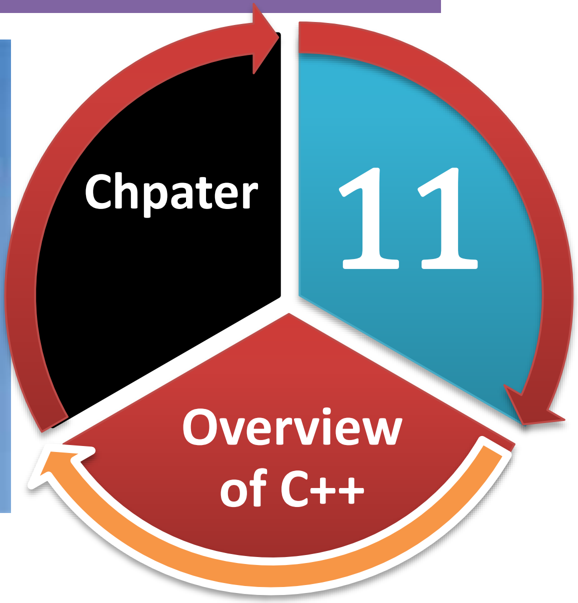
56. What is the difference between early binding and late binding? **Ex-2011**
57. A. what are the differences between member operator function and friend operator functions? **Ex-2014**
58. What are 'unary' and 'binary' operator overloading? **Ex-2013**
59. Why ,if friend function is not used , then left-side argument of binary operator can neither be built-in data type nor be an object of the class other than class having the respective overloaded operator function. **Ex-2013**
60. Give an example of 'unary' operator overloading. **Ex-2013**
61. Is there any way to access private member of a class without taking help of own member function of that class ? Explain your answer with an example.. **Ex-2011**
62. What are multiple , multilevel and hybrid inheritance? **Ex-2011**
63. Explain how 'virtual inheritance ' can solve the problem that is caused when any member of base class may be inherited in different ways to a 'high level derived class'. **Ex-2011**
64. Explain why protected access specifier is used while inheriting a base class? **Ex-2012**
65. Can pointer to a base class be assigned to a pointer to derived class, or, vice versa? **Ex-2013**
66. If both the base class and derived class have a member of same name , which member of the respective class can be and can't be , accessible by the above pointer assignment . **Ex-2013**
67. In derived class constructors and distractors are called in which order? **Ex-2012**
68. Explain the accessing mechanism of data members and member functions in case of Inside main() function and Inside a member function of the same class. **Ex-2013**
69. What is multiple inheritance (virtual inheritance) ? What are its advantages and disadvantages ? **Ex-2012**
70. Briefly discuss inheritance with an example. **Ex-2013**
71. Explain why protected access specifier is used while inheriting a base class? **Ex-2012**
72. In derived class constructors and distractors are called in which order? **Ex-2012**Why is destructor function executed in reverse order of derivation? **Ex-2013**
73. What are the two ways in which a derived class can inherit more than one base class? **Ex-2013**
74. what is wrong with the following definitions:

```
class X
{protected:
      Int a;
 };
 Class Y : public X
 {public :
      Void set (X x, int c)   {x.a=c ;}
  }
```
Ex-2012

75. Implement the following class hieranchy:



76. Explain ,with an example ,what is the rule of 'virtual function' for above situation. **Ex-2013**
77. What is a "virtual member function"? **Ex-2012**
78. What is pure virtual function? What is abstract class? **Ex-2012**
79. What is the difference between virtual function and overloading function? **Ex-2012**
80. What is a pure virtual function ? How a pure virtual function is declared? **Ex-2012**
81. Distinguish between runtime and compile time polymorphism ? How runtime polymorphism is achieved? **Ex-2012**
82. Define early binding and late binding example? **Ex-2012**
83. Explain different types of polumorphism in C++. **Ex-2013**
84. bCreate an abstract base class shape with two members base and height,a member function for initializing and a pure virtual function to compute area().Derive two specific xlasses Triangle and Rectangle Which override the function area().Use these classes in a main function and display the area of a Triangle and a Rectangle. **Ex-2013**
85. Distinguish between static binding and dynamic binding. **Ex-2013**
86. What is 'function template' ? **Ex-2013**
87. Give an example of 'function template'. **Ex-2013**
88. Explain how exception is handling by using 'try' , 'catch' and 'throw' ? **Ex-2013**
89. What's the idea behind templates ? What's the syntax /semantics for a "class template"? **Ex-2012**
90. Define a swap function template for swapping two objects of the same type. **Ex-2012**
91. How can you create a function template in C++? **Ex-2013**
92. Give general syntax of a generic class. **Ex-2013**
93. What are some ways try / catch/ throw can improve software quality ? **Ex-2012**
94. How should I handle resources if my constructors may throw exceptions? **Ex-2012**
95. What is an exception?Why do we need to handle exception? **Ex-2013**
96. what are the keywords used in C++ for executing handling?Describe their usage with suitable example. **Ex-2013**
97. What is steream? Name the streams generally used file I/O. **Ex-2013**
98. Explain various file modes used in C++. **Ex-2013**
99. Explain the various file stream classes needed for file manipulations in C++. **Ex-2013**
100. Explain the functions seekg,seekp, tellg,tellp used for setting pointers during file operation. **Ex-2013**



Previous Question

101. Differentiate between C and C++.When to use C++? **Ex-2013**
102. Write the general program structure of C++ and explain each section **Ex-2013**
103. Explain about different phases for executing a C++ program? **Ex-2013**
104. Write different data types used in C++.What is data abstraction? **Ex-2013**
105. Define Object Oriented programming (OOP). What are features of OOP? **Ex-**2011
106. What is ment by object oriented Programming(OOP)?Write the features of OOP. **Ex-2013**
107. what is object oriented programming? Why is it effective over structured programming? Explain. **Ex-2014**
108. Briefly discuss the properties of oop language **Ex-2014**
109. how can you allocated and free memory to an object? Explain with example? **Ex-2014**
110. How data hiding is accomplished in C++?Explain. **Ex-2013**
111. How are data and functions organized in OOP?Explain. **Ex-2013**
112. What are difference between OOP and Procedural Programming? **Ex-**2011

Question:1.Differentiate between C and C++,When to use C++? Ex-2013 OR
Question:11.What are difference between OOP and Procedural Programming? Ex-2011

Difference between C and C++:

C (PROCEDURAL LANGUAGE)	**C++ (OBJECT ORIENTED)**
C is a subset of C++.	C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.
C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.
C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
In C (because it is a procedural programming language), data and functions are separate and free entities.	In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object.
In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
C does not support function and operator overloading.	C++ supports both function and operator overloading.
C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
C does not provide direct support for error handling (also called exception handling)	C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect.

When to use C++:

CPP use if you want to use advanced algorithms and you know that C++ is a **compiled language**, used for applications for which performance is important, since it usually compiles into byte code, that is binary code, which runs much faster. C++ also offers direct access to chunks of memory and is the lowest-level language of all these three. All the low-end parts of operating systems are programmed in C++ or a similar low-level language, but many apps are as well.

C++ is usually used nowadays when you have the need for low-level, or portable, high performance code. You rarely find it used to implement business logic for enterprise systems, though about 10 years ago you did. You certainly wouldn't want to start out development with an enterprise system with C++ today unless you had a really, really good reason to.

C++ can often be faster than Other programming language when it's optimized properly. It can run with less memory, on more platforms

Question:2 . Write the general program structure of C++ and explain each section Ex-2013

General program structure of C++:

A Sample C++ Program

Let's start with the short sample C++ program shown here.

```
#include <iostream>
using namespace std;
int main()
{
int i;
cout << "This is output.\n"; // this is a single line comment
/* you can still use C style comments */
// input a number using >>
cout << "Enter a number: ";
cin >> i;
// now, output a number using <<
cout << i << " squared is " << i*i << "\n";
return 0;
}
```

Header:

To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>**is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that **<iostream>** is one of the new-style headers defined by Standard C++. New-style headers do not use the **.h** extension. The next line in the program is using namespace std;

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. **A namespace** creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs.

The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library.

Now examine the following line.

int main()

Notice that the parameter list in **main()** is empty. In C++, this indicates that **main()** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here:

int main(void)

This was the way **main()** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

Output:

In C++, the << has an expanded role. It is still the left shift operator, but when it is used as shown in this example, it is also an *output operator*. The word **cout** is an identifier that is linked to the screen. You can use **cout** and the **<<** to output any of the built-in data types, as well as strings of characters.

Input:
cin >> i;
In C++, the **>>** operator still retains its right shift meaning. However, when used as shown, it also is C++'s *input operator*.

Formal way for flashing buffer
Std:: cout<< ::std

Question:4.Write different data types used in C++.What is data abstraction? Ex-2013
Data type in CPP:
C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating	point float
Double	floating point double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

Data abstraction:
Data abstraction and encapuslation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item.
"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." — G. Booch
In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.

Question:5.Define Object Oriented programming (OOP). What are features of OOP? Ex-2011,Ex-2013
Object Oriented Programming:
OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs.
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred

instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

Feature of CPP:

C++ has Three features

1. **Encapsulation:**
2. **Polymorphism**
3. **Inheritance**

Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created.

Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Data encapsulation led to the important OOP concept of data hiding.

Data Encapsulation Example:

```
#include <iostream>
using namespace std;

class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
       total += number;
    }
    // interface to outside world
    int getTotal()
    {
       return total;
    };
  private:
    // hidden data from outside world
    int total;
};
```

```
int main( )
{
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result: Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Polymorphism:

Object-oriented programming languages support polymorphism, which is characterized by the phrase "one interface, multiple methods." **In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.**

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance

A real-word example of polymorphism is a thermostat. No matter what type of furnace your hou has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (meth you have. For example, if you want a 70-degree temperature, you set the thermostat 70 degrees. It doesn't matter what type of furnace actually provides the heat. **C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.**

Consider the following example where a base class has been derived by other two classes:

```
#include <iostream>
using namespace std;

class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape{
  public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape{
  public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
Shape *shape;
Rectangle rec(10,7);
Triangle  tri(10,5);

// store the address of Rectangle
shape = &rec;
// call rectangle area.
shape->area();

// store the address of Triangle
shape = &tri;
// call triangle area.
shape->area();

return 0;
}
```

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

```
Parent class area
Parent class area
```

Inheritance:

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications.

For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an important aspect of object-oriented programming.
When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Question:6.What is object oriented programming? Why is it effective over structured programming? Explain. Ex-2014

Object Oriented Programming:

OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

Effective over structure programming:

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected).

Using only structured programming techniques, **programs are typically organized around code.** This approach can be thought of as "code acting on data." For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program.

Object-oriented programs work the other way around. **They are organized around data, with the key principle being "data controlling access to code."** In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

Question:7.Briefly discuss the properties of oop language Ex-2014

Properties of OOP:

Encapsulation – Encapsulation is capturing data and keeping it safely and securely from outside interfaces.

Inheritance- This is the process by which a class can be derived from a base class with all features of base class and some of its own. This increases code reusability.

Polymorphism- This is the ability to exist in various forms. For example an operator can be overloaded so as to add two integer numbers and two floats.

Abstraction- The ability to represent data at a very conceptual level without any details.

Question:How data hiding is accomplished in C++?Explain. Ex-2013

Data hiding in CPP:

Data encapsulation led to the important OOP concept of data hiding. Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We know that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:

```
class Box
{
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

Data Hiding Example:

```
#include <iostream>
using namespace std;

class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
       total += number;
    }
    // interface to outside world
```

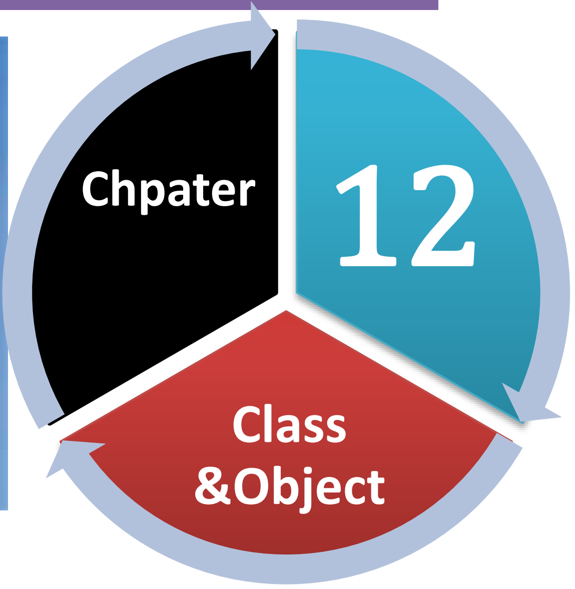
```
      int getTotal()
      {
         return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Question:10.How are data and functions organized in OOP?Explain. Ex-2013



Important Questions

1. What are 'class' and 'object'? **Ex-2011**
2. what is friend function? Why ist it used ? describe with and example? **Ex-2014**
3. What is a friend? Do friends violate encapsulation? **Ex-2012**
4. What are some advantages / disadvantages of using friend functions? **Ex-2012**
5. what is in-line function ? what are the restrictions to in-line functions? **Ex-2014**
6. What is inline function? **Ex-2012**
7. How do you tell the compiler to make a member function inline? **Ex-2012**
8. How do you tell the compiler to make a member function inline? **Ex-2012**
9. What are advantages of using 'static member variable' and 'static member function' of a class? **Ex-2011**
10. Give an example of 'nested class member'. **Ex-2011**
11. Define constructor and destructor?
12. what are constructor and destructor function? Can these functions have input parameters
13. and return type? **Ex-2014**
14. Explain the accessing mechanism of data members and member functions in case of
15. Inside main() function and Inside a member function of the same class. **Ex-2013**
16. When do you declare a member of a class static

Question:1.What are 'class' and 'object'? Ex-2011

Class:

Class is a collection of data member and member function. Class is a user define data type

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class. A class declaration is similar syntactically to a structure. In Chapter 11, a simplified general form of a class declaration was shown. Here is the entire general form of a **class** declaration that does not inherit any other class.

```
class class-name {
private data and functions
access-specifier:
data and functions
access-specifier:
data and functions
// ...
access-specifier:
data and functions
} object-list;
```

The *object-list* is optional. If present, it declares objects of the class. Here,*access-specifier* is one of these three C++ keywords:

public
private
protected

Object:

Object is a class type variable. **Objects** are also called instance of the class. Each **object** contains all members(variables and functions) declared in the class.

Question: What is Class Access specifier Explaine with example? Or
Question: Explain the accessing mechanism of data members and member functions in case of Inside main() function and Inside a member function of the same class. Ex-2013

Class specifier:

Access-specifier is one of these three C++ keywords:

public
private
protected

Private:

Functions and data declared within a class are private to that class and may be accessed only by other members of the class.**A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.**

By default all the members of a class would be private, for example in the following class **width** is a private member, which means until you label a member, it will be assumed a private member:

```
class Box
```

```
{
  double width;
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );
};
```

Accessing Private Data Members
To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member. The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

```
Example :
class Student
{
 private:   // private data member
 int rollno;

 public:    // public accessor and mutator functions
 int getRollno()
 {
  return rollno;
 }

 void setRollno(int i)
 {
  rollno=i;
 }

};

int main()
{
 Student A;
 A.rollono=1;  //Compile time error
 cout<< A.rollno; //Compile time error

 A.setRollno(1);  //Rollno initialized to 1
 cout<< A.getRollno(); //Output will be 1
}
```

Public:
The public access specifier allows functions or data to be accessible to other parts of your program.A **public member is accessible from anywhere outside the class but within a program.** You can set and get the value of public variables without any member function

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student
{
 public:
 int rollno;
 string name;
};

int main()
{
 Student A;
A.rollno=1;
 A.name="Adam";
 cout <<"Name and Roll no of A is :"<< A.rollno;

}
```

Protected:

The protected access specifier is needed only when inheritance is involved . Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached. A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes

Accessing Protected Data Members

Protected data members, can be accessed directly using dot (.) operator inside the subclass of the current class, for non-subclass we will have to follow the steps same as to access private data member.

Question:2.Define constructor and destructor?

Constructor: A class constructor is a special function in a class that is called when a new object of the class is created.

A constructor function is a special function that is a member of a class and has the same name as that class. For example, here is how the **stack** class looks when converted to use a constructor function for initialization:

// This creates the class stack.

```
class stack {
int stck[SIZE];
int tos;
public:

stack(); // constructor
void push(int i);
int pop();

};
```

Notice that the constructor **stack()** has no return type specified. In C++, constructor functions cannot return values and, thus, have no return type.

The **stack()** function is coded like this:

```
// stack's constructor function
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";

}
```

Destructo: A destructor is also a special function which is called when created object is deleted.

Destructor is a function that has the same name as that of the class but is prefixed with a ~(tilde). Destructors de-initialize Object when they are destroyed. You can think of destructors as the opposite of constructors: constructors execute when objects are created, and destructors execute when the objects are destroyed by the built in Garbage Collection facility. This process occurs behind the scenes with no consequence to the programmer.

The complement of the constructor is the *destructor*. In many circumstances, anobject will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called

```
  public class Car {
int start; //constructor
public Car()
{ start = 0; }
//destructor
~Car() { start = 0; } }
```

Question:3. What are constructor and destructor function? Can these functions have input parameters and return type? Ex-2014

<u>Can these functions have input parameters and return type:</u>

A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used (e.g. open a file or database).

Unlike normal member functions, constructors have specific rules for how they must be named:

1. Constructors should always have the same name as the class (with the same capitalization)
2. Constructors have no return type (not even void)

Note that constructors are only used for initialization. They can not be explicitly called.

Question:What is parameterized constructor? Explain with example?

Parameterized Constructor:

It is possible to pass arguments to constructor functions. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Example:

```
#include <iostream>
    using namespace std;
    class myclass {
                        int a, b;
                public:
                        myclass(int i, int j) {a=i; b=j;}
                        void show() {cout << a << " " << b;}
              };
    int main()
    {
                        myclass ob(3, 5);
                        ob.show();
    return 0;
    }
```

Question:3. How Structures and Classes Are Related? Different between structure and classes?

Different Structure and Classes

Class	**Structure**
Members of a class are private by default	members of struct are public by default.

class Test {
  int x; // x is private
};
int main()
{
 Test t;
 t.x = 20; // compiler error because x is private
 getchar();
 return 0;
}
	struct Test {
	  int x; // x is public
	};
	int main()
	{
	 Test t;
	 t.x = 20; // works fine because x is public
	 getchar();
	 return 0;
	}

when deriving a class, default access specifier is private.	When deriving a struct from a class/struct, default access-specifier for a base class/struct is public.

class Base {
public:
  int x;
};
	class Base {
	public:
	  int x;
	};
	
	struct Derived : Base { }; // is equilalent to struct

class Derived : Base { }; // is equilalent to class Derived : private Base {}
 	
int main()
{
  Derived d;
  d.x = 20; // compiler error becuase inheritance is private
  getchar();
  return 0;
}
	Derived : public Base {}
	
	int main()
	{
	  Derived d;
	  d.x = 20; // works fine becuase inheritance is public
	  getchar();
	  return 0;
	}

Question:4.what is friend function? Why ist it used ? describe with and example? Ex-2014

Friend function:

A friend function that is a "friend" of a given class is allowed access to private and protected data in that class that it would not normally be able to as if the data was public .

Why ist it used:

Normally, A **function** that is defined outside of a class cannot access such information. It is possible to grant a nonmember function access to the private members of a class by using a **friend.**

A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend.** To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend.**

Friend Function Example:

Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
        int a, b;
public:
        friend int sum(myclass x);
        void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
        // Note: sum() is not a member function of any class.
int sum(myclass x)
{
        /* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

<u>Question:5.</u>What is friend class describe with example?

<u>Friend class:</u>

<u>Friend class Example:</u>

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

<u>Example of a friend class:</u>

```
    #include <iostream>
    using namespace std;
class TwoValues {
        int a;
         Int b;
    public:
    TwoValues(int i, int j) { a = i; b = j; }
     friend class Min;
       };

    class Min {
        public:
            int min(TwoValues x);
          };
    int min(TwoValues x)
       {
        return x.a < x.b ? x.a : x.b;
        }
    int main()
    {
    TwoValues ob(10, 20);
    Min m;
    cout << m.min(ob);
      return 0;
    }
```

class Min has access to the private variables a and b declared within the TwoValues class. It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class

Question:6. Describe some Advantage and disadvantage of using friend function?

<u>A friend function has the following advantages :</u>

- Provides additional functionality which is kept outside the class
- Provides functions that need data which is not normally used by the class..
- Allows sharing private class information by a non member function..
- They provide a degree of freedom in the interface design options.
- We can able to access the other class members in our class if,we use friend keyword.
- We can access the members without inheriting the class.

Disadvantages

- The major disadvantage of friend functions is that they require an extra line of code when you want dynamic binding.
- A derived class can't inherit friend functions.
- Since the friend function can access to the data members and member functions which may be either private or public of any class remaining outside of the class so it can break the security.

Question:6.What is a friend? Do friends violate encapsulation? Ex-2012

Friend violet encaptulation:

No! If they're used properly, they enhance encapsulation."Friend" is an explicit mechanism for granting access, just like membership. You cannot (in a standard conforming program) grant yourself access to a class without modifying its source.

For example:

```
class X {
   int i;
public:
   void m();      // grant X::m() access
   friend void f(X&);  // grant f(X&) access
   // ...
};

void X::m() { i++; /* X::m() can access X::i */ }

void f(X& x) { x.i++; /* f(X&) can access X::i */ }
```

if you use **friend** functions as a syntactic variant of a class's public access functions, they don't **violate encapsulation** any more than a member function **violates encapsulation**.

Question:What is in-line function ? what are the restrictions to in-line functions? Ex-2014 ,Ex-2012

Inline function:

C++ **inline function** is powerful concept that is commonly used with classes. If a **function** is **inline**, the compiler places a copy of the code of that **function** at each point where the **function** is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. **The compiler can ignore the inline qualifier in case defined function is more than a line.**

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Inline Function Example:

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;
  int main()
   {
     cout << (10>20 ? 10 : 20);
      cout << " " << (99>88 ? 99 : 88);
    }
```

Inline Class Member Example:

inline functions may be class member functions. For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;
class myclass {
                          int a, b;
              public:
                          void init(int i, int j);
                          void show();
               };

// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
// Create another inline function.
inline void myclass::show()
{
cout << a << " " << b << "\n";
}
```

```
int main()
{
myclass x;
        x.init(10, 20);
        x.show();
return 0;
}
```

Restriction of inline function:

1) May increase function size so that it may not fit on the cache, causing lots of cahce miss.
2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if some body changes code inside inline function than all calling location will also be compiled.
4) If used in header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
6) Its not useful for embeded system where large binary size is not preferred at all due to memory size constraints.

Question: How do you tell the compiler to make a member function inline? Ex-2012

Make member function inline:

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line. A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Example:

```
#include <iostream>
using namespace std;
class myclass {
                int a, b;
        public:
                void init(int i, int j);
                void show();
        };

// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
// Create another inline function.
inline void myclass::show()
{
cout << a << " " << b << "\n";
}

int main()
{
        myclass x;
                x.init(10, 20);
                x.show();
        return 0;
}
```

When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword.

Question:why might the following function not be in-lined by your compiler?

```
                Void f1( )
            i.  {
                int i ;
                for(i=0;i<10;i++)  cout<< i ;
                }  Ex-2012
```

Answer:

This function not be in-line because not have any argument ,

Question:What is static class member explain with example?

Static Class Member:

Both function and data members of a class can be made static. This section explains the consequences of each.

Static data member:

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable All static variables are initialized to zero before the first object is created.

When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs.

To understand the usage and effect of a static data member, consider this program:

```
 #include <iostream>
using namespace std;
class shared {
            static int a;
             int b;
      public:
            void set(int i, int j) {a=i; b=j;}
             void show();
} ;
int shared::a; // define a
void shared::show()
{
                cout << "This is static a: " << a;
                cout << "\nThis is non-static b: " << b;
                cout << "\n";
```

```
        }
            int main()
            {
                        shared x, y;
                        x.set(1, 1); // set a to 1
                        x.show();
                        y.set(2, 2); // change a to 2
                        y.show();
                        x.show(); /* Here, a has been changed for both x and y
                        because a is shared by both objects. */
            return 0;
            }
```

his program displays the following output when run.

```
static a: 1
non-static b: 1
static a: 2
non-static b: 2
static a: 2
non-static b: 1
```

Static member function:

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class. (Of course, global functions and data may be accessed by static member functions.) A static member function does not have a this pointer.

There cannot be a static and a non-static version of the same function. A static member function may not be virtual. Finally, they cannot be declared as const or volatile.

For example that get_resource( ) is now declared as static.

illustrates, get_resource( ) may be called either by itself, independent of any object, by using the class name and the scope resolution operator, or in connection with an object.

```
#include <iostream>
using namespace std;
class cl {
          static int resource;
 public:
          static int get_resourc ();
           void free_resource() { resource = 0; }
};
int cl::resource; // define resource
int cl::get_resource()
```

```
            {
                if(resource) return 0;  // resource already in use
            else {
                resource = 1;
                return 1; // resource allocated to this object
            }
            }
int main()
{
cl ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
    if(cl::get_resource()) cout << "ob1 has resource\n";
    if(!cl::get_resource()) cout << "ob2 denied resource\n";
        ob1.free_resource();
        if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
return 0;
}
```

Question:When do you declare a member of a class static?

Answer:

When the same data needs to be accessible between multiple instances of a class.

```
class Salesman
            {
              public static int numOfSalesmen;
                  public Salesman()
                      { ... numOfSalesmen++; ...
                      } ... }

Salesman a;
Salesman b;
Salesman c;

a.numOfSalesmen,
b.numOfSalesmen,
c.numOfSalesmen, and
 Salesman.numOfSalesmen are all 3 now
```

- In C++, a static class member is common to all instances of that class. A static class method is a method that can only access static class members. Non static methods can also access static members, but remember that there is only one instance of that member amongst all instances of the class. One advantage of a static method is that you do not have to instantiate the class to use the static member or static method.

Question:What are advantages of using 'static member variable' and 'static member function' of a class? Ex-2011

Advantage of static member variable

- When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object.
- All **static** variables are initialized to zero before the first object is created.
- Persistence: it remains in memory until the end of the program.
- File scope: it can be seen only within a file where it's defined.
- Visibility: if it is defined within a function/block, it's scope is limited to the function/block. It cannot be accessed outside of the function/block.
- A **static** member variable exists before any object of its class is created.

Advantage of static member function:

Static member function can only access static member data, or other static member functions while non-static member functions can access all data members of the class: static and non-static.

Restriction of static member function:

There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.)

- A **static** member function does not have a **this** pointer.
- There cannot be a **static** and a non-**static** version of the same function.
- A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**.

Question: When Constructor and destructor are Execution?

Constructor and Destructor Execute:

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

A local object's constructor function is executed when the object's declaration statement is encountered. The destructor functions for local objects are executed in the reverse order of the constructor functions.

Global objects have their constructor functions execute before main( ) begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order after main( ) has terminated.

Example:

```
#include <iostream>
using namespace std;
        class myclass {
                public:
int who;
myclass(int id);
~myclass();
} glob_ob1(1), glob_ob2(2);
myclass::myclass(int id)
     {
          cout << "Initializing " << id << "\n";
          who = id;
}
 myclass::~myclass()
     {
          cout << "Destructing " << who << "\n";
}
int main()
{
myclass local_ob1(3);
cout << "This will not be first line displayed.\n";
myclass local_ob2(4);
return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
```

Destructing 4

Question: What is scope resolution? Explain with example?

The scope resolution operator:

The :: operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

For example, consider this fragment:

```
int i; // global i
void f()
{
int i; // local i
i = 10; // uses local i
.
.
.
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f()** needs to access the global version of **i**? It may do so by receding the **I** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
int i; // local i
::i = 10; // now refers to global i
.
.
.
}
```

Question:Give an example of 'nested class member'. Ex-2011

Nested class:

It is possible to define one class within another. Doing so creates a nested class. Since a class declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, nested classes are seldom used.

Explanation

The name of the nested class exists in the scope of the enclosing class, and name lookup from a member function of a nested class visits the scope of the enclosing class after examining the scope of the nested class. As any member of its enclosing class, nested class has access to all names (private, protected, etc) to which the enclosing class has access, but it is otherwise independent and has no special access to the **this** pointer of the enclosing class.

Example of nested class:

Nested classes can be forward-declared and later defined, either within the same enclosing class body, or outside of it:

```
class enclose {
   class nested1; // forward declaration
   class nested2; // forward declaration
   class nested1 {}; // definition of nested class
};
class enclose::nested2 { }; // definition of nested class
```

Question: how object passing to a function give example?

Question: is it possible passing object to a function? How explain with example.

Passing object to function

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism. This means that a copy of an object is made when it is passed to a function. However, the fact that a copy is created means, in essence, that another
object is created.

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2);
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying
```

Question: how a Object return? explain with example? When an object is returned by a function then what happened?

<u>Object return:</u>

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

Object is returned by a function then what happened:

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it.

Question: Explain how can object assignment ? Give with Example.

Object assignment bit by bit

Assuming that both objects are of the same type, you can assign one object to another. This causes the data of the object on the right side to be copied into the data of the object on the left. For example, this program displays **99**:

// Assigning objects.

```
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
int main()
{
myclass ob1, ob2;
ob1.set_i(99);
ob2 = ob1; // assign data from ob1 to ob2
cout << "This is ob2's i: " << ob2.get_i();
return 0;
}
```

By default, all data from one object is assigned to the other by use of a bit-by-bit copy.

Question:Find the error(s) in each of the following and explain how to correct it:

i. **void ~Time(int);**

ii. **The following is a partial definition of class time:**

```
class Time{
public:
// function prototypes
Private:
 int hour=0;
 int minute=0;
 int second=0;
```

iii. Assume the following prototypes is declared in class Employee:

int Employee (const char * , const char *); Ex-2012

Answer:

(i): Destructor function **Time** return type does not allowed.

(ii). Class member variable does not allowed assigning value. Or initialization not possible in class member variable. Initialization must be into a function.

(iii) employee is a constructor function but its not allowd any return type it's a class type special function.

Question:Find the errors in the following class and explain how to correct them:

```
class Example{
public:
Example (int y=10)
  : data(y)
{
   //empty body
}
int get Incremented Data ( ) const
{
  Return ++ data;
}
Static int get Count( )
{
  cout<< "Data is"<<data<<end1 ;
  return count;
}
Private:
  int data;
  static int count ;
}; // end class Example
```

Ex-2012

Answer:

Question:Given the following partial class , add the necessary constructor functions so that both 4 declarations within main ( ) are valid.

```
Class samp {
   int a;
public:
    //add constructor functions

    int get _a( ) {return a ; }
};
int main( ) {
      samp ob (88); // init ob's a to 88
      samp ob array [10]; //noninitialized 10-element array//....
}
```

Answer:

```
           Class samp {
   int a;
public:
samp(int k) {k=a;}

    int get _a( ) {return a ; }
};
int main( ) {
      samp ob (88); // init ob's a to 88
      samp ob array [10]; //noninitialized 10-element array//....
}
```

1. What is the difference between the effects of the following two declarations:

Ratio y(x) ;

Ratio y=x ; **Ex-2012**

2. What is the difference between the effects of the following two lines:

Ratio y=x ;

Ratio y; y=x ; **Ex-2012**

Important Questions

1. Can an array of objects be initialized?Explain. **Ex-2013**
2. How are memory allocated for different type of members of a class? **Ex-2011**
3. What is “this” pointer? **Ex-2012**
4. What is “this”?Explain with an example. **Ex-2013**
5. What is new and delete?What are their advantages? **Ex-2013**
6. What is reference?Describe with example. **Ex-2013**
7. Can an array of objects be initialized?Explain.
8. What is “this”?Explain with an example. **Ex-2013**
9. What is new and delete?What are their advantages? **Ex-2013**
10. What is reference?Describe with example. **Ex-2013**
11. Write the output for the following lines

```
std::cout <<i << ‘\n’ ;
std::cout << “i” << ‘\n’ ;
std::cout << i /j << ‘\n’ ;
std::cout <<”i=” << i ;
```

Question:What is Array of objects ? Explain with example.

Arrays of Objects:

The syntax for declaring and using an object array is exactly the same as it is for any other type of array. **For example**, this program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
    return 0;
}
```

Question:Can an array of objects be initialized?Explain. Ex-2013

Answer:

Yes we can initialized an array of objects. If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructor function. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax.

For example, here is a slightly different version of the preceding program that uses an initialization

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen. Actually, the initialization syntax shown in the preceding program is shorthand for this longer form:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Here, the constructor for **cl** is invoked explicitly.

Or

```
class cl {
int i;
public:
cl() { i=0; } // called for non-initialized arrays
cl(int j) { i=j; } // called for initialized arrays
int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized
cl a2[34]; // uninitialized
```

Question: What is Pointers to Object ?Explain with example?

<u>Pointers to Objects:</u>

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

```
#include <iostream>
using namespace std;

class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer.

For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

```
#include <iostream>
using namespace std;
class cl {
int i;
```

```
public:
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3};
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int j) { i=j; }
};
int main()
{
cl ob(1);
int *p;
p = &ob.i; // get address of ob.i
cout << *p; // access ob.i via p
return 0;
}
}
```

Question: what are the different between Intialized and uninitialized array?

<u>Creating Initialized vs. Uninitialized Arrays:</u>

A special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
```

Here, the constructor function defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration:

```
cl a[9];
      error constructor requires initializers
```

To solve this problem, you need to overload the constructor function, adding one that takes no parameters. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
int i;
public:
cl() { i=0; } // called for non-initialized arrays
cl(int j) { i=j; } // called for initialized arrays
int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized
cl a2[34]; // uninitialized
```

Question:What is "this" pointer? Ex-2012 or
Question:What is "this"?Explain with an example. Ex-2013
.
This pointer:

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called),This pointer is called **this.**

Or

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

To understand **this**, first consider a program that creates a class called **mul** that computes the result of a number raised to some power:

```
#include <iostream>
using namespace std;
class mul {
int a,b;
public:
    p(int x,int y);
    double get_value() { return val; }
};
p::p(int x, int y)
{
a = x;
b = y;
val = 1;
val=x*y;
int main()
{
P  x(4.0, 2)
cout << x.get_value() << " ";
return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **mul(),** the statement a=x;; means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

```
this->a = x
```

The **this** pointer points to the object that invoked **sum().** Thus, **this –>a** refers to that object's copy of **a.**

Here is the entire **mul()** function written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
this->a = x;
this->b = y;
this->val = 1;
this->val = this->a * this->b;
}
```

Remember that the **this** pointer is automatically passed to all member functions. Therefore, **get_value()** could also be rewritten as shown here:

```
double get_value() { return this->val; }
```

In this case, if **get_value()** is invoked like this:

```
x.get_value();
```

then **this** will point to object **x.**

Two final points about **this.** First, **friend** functions are not members of a class and, therefore, are not passed a **this** pointer. Second, **static** member functions do not have a **this** pointer.

Question:Can pointer to a base class be assigned to a pointer to derived class, or, vice versa? Ex-2013

Question: If both the base class and derived class have a member of same name , which member of the respective class can be and can't be , accessible by the above pointer assignment . Ex-2013

Question:What the condition if a pointer want to point a derives types object Explain with example?

<u>Pointers to Derived Types:</u>

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B *** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D *** may not point to an object of type **B.**

Here is a short program that illustrates the use of a base pointer to access derived objects.

```
#include <iostream>
using namespace std;
class base {
int i;
public:
void set_i(int num) { i=num; }
int get_i() { return i; }
};
class derived: public base {
int j;
public:
void set_j(int num) { j=num; }
int get_j() { return j; }
};
int main()
{
base *bp;
derived d;
bp = &d; // base pointer points to derived object
// access derived object using base pointer
bp->set_i(10);
cout << bp->get_i() << " ";
/* The following won't work. You can't access element of
a derived class using a base class pointer.
bp->set_j(88); // error
cout << bp->get_j(); // error
*/
return 0;
}
```

As you can see, a base pointer is used to access an object of a derived class. Although you must be careful, it is possible to cast a base pointer into a pointer of the derived type to access a member of the derived class through the base pointer. For example, this is valid C++ code:

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

It is important to remember that pointer arithmetic is relative to the base type of the pointer.

Question:What is reference?Describe with example. Ex-2013

References:

C++ contains a feature that is related to the pointer called a reference. A reference is essentially an implicit pointer.

There are three ways that a reference can be used:

- as a function parameter,
- as a function return value, or
- as a stand-alone reference

Reference Parameters:

The most important use for a reference is to allow you to create functions that Automatically use call-by-reference parameter passing.

Arguments can be passed to functions in one of two ways:

- Using call-by-value or
- Call-by-reference.

Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing.

- First, you can explicitly pass a pointer to the argument.
- Second, you Call-by-reference passes the address of the argument to the function.

Example:

To create a reference parameter, precede the parameter's name with an &. For example, here is how to declare neg() with i declared as a reference parameter:

```
void neg(int &i);
```

```
// Use a reference parameter.
#include <iostream>
using namespace std;
void neg(int &i); // i now a reference
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(x); // no longer need the & operator
cout << x << "\n";
return 0;
}
void neg(int &i)
{
i = -i; // i is now a reference, don't need *
}
```

Here is another example. This program uses reference parameters to swap the values of the variables it is called with. The swap() function is the classic example of call-by-reference parameter passing.

```
#include <iostream>
using namespace std;
void swap1(int  i, int  j);
void swap2(int &i, int &j);

int main()
{
int a, b, c, d;
a = 1;
b = 2;
c = 3;
d = 4;
cout << "a and b: " << a << " " << b << "\n";
swap1(a, b); // no & operator needed
cout << "a and b: " << a << " " << b << "\n";

cout << "c and d: " << c << " " << d << "\n";
swap2(c, d);
cout << "c and d: " << c << " " << d << "\n";
return 0;
}
void swap1(int  i, int  j)
{
int t;
t = i; // no * operator needed
i = j;
j = t;
}
void swap2(int &i, int &j)
{
int t;
t = i; // no * operator needed
i = j;
j = t;
}
```

This program displays the following:

```
a and b: 1 2
a and b: 1 2

c and d: 3 4
c and d: 4 3
```

Question:How Passing References to Objects Explain with example?

Passing References to Objects

When an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. If for some reason you do not want the destructor function to be called, simply pass the object by reference.

When you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called. For example, try this program.

```
#include <iostream>
using namespace std;
class cl {
int id;
public:
int i;
cl(int i);
~cl();
void neg(cl &o) { o.i = -o.i; } // no temporary created
};
cl::cl(int num)
{
cout << "Constructing " << num << "\n";
id = num;
}
cl::~cl()
{
cout << "Destructing " << id << "\n";
}
int main()
{
cl o(1);
o.i = 10;
o.neg(o);
cout << o.i << "\n";
return 0;
}
```

Here is the output of this program:

```
Constructing 1
-10
Destructing 1
```

As you can see, only one call is made to cl's destructor function. Had o been passed by value, a second object would have been created inside neg(), and the destructor would have been called a second time when that object was destroyed at the time neg() terminated. As the code inside neg() illustrates, when you access a member of a class through a reference, you use the dot operator. The arrow operator is reserved for use with
pointers only.

Returning References:

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement! For example, consider this simple program.

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

This program replaces the space between Hello and There with an X. That is, the program displays HelloXthere. Take a look at how this is accomplished.

First, replace() is declared as returning a reference to a character. As replace() is coded, it returns a reference to the element of s that is specified by its argument i. The reference returned by replace() is then used in main() to assign to that element the character X.

Question:What is independendent reference? Give a example.

Independent References:

By far the most common uses for references are to pass an argument using call-by-reference and to act as a return value from a function. However, you can declare a reference that is simply a variable. This type of reference is called an independent reference.

When you create an independent reference, all you are creating is another name for an object variable. All independent references must be initialized when they are created. The reason for this is easy to understand. Aside from initialization, you cannot change what object a reference variable points to. Therefore, it must be initialized when it is declared. (In C++, initialization is a wholly separate operation from assignment.)

The following program illustrates an independent reference:
Chapter 13: Arrays, Pointers, References, and the Dynamic Allocation Operators 347

```
#include <iostream>
using namespace std;
int main()
{
int a;
int &ref = a; // independent reference
a = 10;
cout << a << " " << ref << "\n";
ref = 100;
cout << a << " " << ref << "\n";
int b = 19;
ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref--; // this decrements a
// it does not affect what ref refers to
cout << a << " " << ref << "\n";
return 0;
}
```

The program displays this output:

```
10 10
100 100
19 19
18 18
```

Actually, independent references are of little real value because each one is, literally, just another name for another variable. Having two names to describe the same object is likely to confuse, not organize, your program.

Question: How the process of reference to derived types? What are the restriction to reference?

References to Derived Types:

Similar to the situation as described for pointers earlier, a base class reference can be used to refer to an object of a derived class. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions to References:

There are a number of restrictions that apply to references. You cannot reference another reference. Put differently, you cannot obtain the address of a reference. You cannot create arrays of references. You cannot create a pointer to a reference. You cannot reference a bit-field. A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value. Null references are prohibited.

Question:How are memory allocated for different type of members of a class? Ex-2011

Answer:

Not every variable in a program has a permanently assigned area of memory, instead, modern languages are smart about giving memory to a variable only when necessary. When we use the term **allocate**, we indicate that the variable is given an area of memory to store its value. A variable is **deallocated** when the system reclaims the memory from the variable, so it no longer has an area to store its value.

For a variable, the period of time from its allocation until its deallocation is called its **lifetime**. The most common memory related error is using a deallocated variable. For **local variables**, modern languages automatically protect against this error. In other words, most of the time, the local variables appear automatically when we need them, and they disappear automatically when we are done with them. With **pointers**, however, programmers must make sure that allocation is handled correctly.

Example:

```
#include <iostream>
#include <string>
using namespace std;

class Student
{
   float gpa;
   public:
   void read()
   {
      cin>>gpa;
   }

   void display()
   {
      cout<<"STUDENT GPA : "<<GPA<<endl;
   }
};

void main()
{
   Student s1;
}
```

For the object s1, 4 bytes must be allocated in main memory for float variable. constructors and destructor should not need any space in the object itself.

Question: What are the Dynamic Allocation operators?

C++'s Dynamic Allocation Operators:

C++ provides two dynamic allocation operators: new and delete. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs. As explained in Part One, C++ also supports dynamic memory allocation functions, called malloc() and free(). These are included for the sake of compatibility with C. However, for C++ code, you should use the new and delete operators because they have several advantages.

The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new. The general forms of new and delete are shown here:

p_var = new type; delete p_var;

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type. Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then new will fail and a bad_alloc exception will be generated. This exception is defined in the header <new>. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

Question:How can you allocated and free memory to an object? Explain with example? Ex-2014 or

Question:What is new and delete?What are their advantages? Ex-2013

C++ provides two dynamic allocation operators: new and delete. These operators are used to allocate and free memory at run time.

<u>New :</u> The new operator allocates memory and returns a pointer to the start of it.
The general forms of new is:

p_var = new type;

<u>Delete</u>:
The delete operator frees memory previously allocated using new. The general forms of delete is shown here:

delete p_var;

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type. Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then new will fail and a bad_alloc exception will be generated. This exception is defined in the header <new>. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

Question: What is nothrow Alternative explain with example?

<u>The nothrow Alternative:</u>
it is possible to have new return null instead of throwing an exception when an allocation failure occurs. This form of new is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to malloc() with new. (This is common when updating C code to C++.) This form of new is shown here:

p_var = new(nothrow) type;

Here, p_var is a pointer variable of type. The nothrow form of new works like the original version of new from years ago. Since it returns null on failure, it can be "dropped into" older code without having to add exception handling. However, for new code, exceptions provide a better alternative. To use the nothrow option, you must include the header <new>.

The following program shows how to use the new(nothrow) alternative.

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p, i;
p = new(nothrow) int[32]; // use nothrow option
if(!p) {
cout << "Allocation failure.\n";

return 1;
}
for(i=0; i<32; i++) p[i] = i;
for(i=0; i<32; i++) cout << p[i] << " ";
delete [] p; // free the memory
return 0;
}
```

As this program demonstrates, when using the nothrow approach, you must check the pointer returned by new after each allocation request.

Question:in c++ there are two ways to achieve call by reference . what are they? Discuss with example. Ex-2014

Question:Write the output for the following lines

```
std::cout <<i << '\n' ;
std::cout << "i" << '\n' ;
std::cout << i /j << '\n' ;
std::cout <<"i=" << i ;
```

Important Questions

2. What is copy constructor? When copy constructor called?
3. What is default argument? **Ex-2011**
4. in c++ there are two ways to achieve call by reference . what are they? Discuss with example. **Ex-2014**
5. What is inline function? **Ex-2012**

Question: What is Function overloading ?Explain with example.

Function overloading:

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters.

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

It is only through these differences that the compiler knows which function to call in any given situation.

Differ from each other by the types

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

Differ from each other by the number of arguments:

The next program overloads **myfunc()** using a different number of parameters:

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
int myfunc(int i)
{
```

```
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt
to overload **myfunc():**

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Question:is it possible to overloaded constructor function? If possible explain with example.

<u>Overloading Constructor Functions:</u>

Constructor functions can be overloaded; in fact, overloaded constructors are very common.

There are three main reasons why you will want to overload a constructor function:

- **To gain flexibility,**
- **To allow both initialized and uninitialized objects to be created, and**
- **To define copy constructors overloading a constructor to gain flexibility**

Many times you will create a class for which there are two or more possible ways to construct an object. In these cases, you will want to provide an overloaded constructor function for each way. This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

The user is free to choose the best way to construct an object given the specific circumstance. Consider
this program that creates a class called **date**, which holds a calendar date.

```
#include <iostream>
#include <cstdio>
using namespace std;
class date {
int day, month, year;
public:
date(char *d);
date(int m, int d, int y);
void show_date();
};
// Initialize using string.
date::date(char *d)
{
scanf(d, "c%d%*c%d%*c%d", &month, &day, &year);
}
// Initialize using integers.
date::date(int m, int d, int y)
{
day = d;
month = m;
year = y;
}
void date::show_date()
{
cout << month << "/" << day;
cout << "/" << year << "\n";
}
int main()
{
date ob1(12, 4, 2001), ob2("10/22/2001");
ob1.show_date();
ob2.show_date();
return 0;
}
```

Question:What is copy constructor? When copy constructor called?

Copy constructor:

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
  // body of constructor
}
```

When copy constructor called:

when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created.

For example, assume a class called MyClass that allocates memory for each object when it is created, and an object A of that class. This means that A has already allocated its memory. Further, assume that A is used to initialize B, as shown here:

```
MyClass B= A;
```

If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own. Clearly, this is not the desired outcome.

For example, if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed! The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances. To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. When a copy constructor exists, the default, bitwise copy is bypassed.

The most common generalform of a copy constructor is

```
classname (const classname &o) {
// body of constructor
}
```

Here, o is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them.
The first is assignment.

The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y); // y passed as a parameter
y = func(); // y receiving a temporary, return object
```

Example of copy constructor:

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
class array {
int *p;
int size;
public:
array(int sz) {
try {
p = new int[sz];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
size = sz;
}
~array() { delete [] p; } // copy constructor
array(const array &a);
void put(int i, int j) {
if(i>=0 && i<size) p[i] = j;
}
int get(int i) {
return p[i];
}
};
// Copy Constructor
array::array(const array &a) {
int i;
try {
p = new int[a.size];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
for(i=0; i<a.size; i++) p[i] = a.p[i];
}
int main()
{
array num(10);
int i;
for(i=0; i<10; i++) num.put(i, i);
for(i=9; i>=0; i--) cout << num.get(i);
cout << "\n";  // create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);
return 0;
```

}

Let's look closely at what happens when **num** is used to initialize **x** in the statement array x(num); // invokes copy constructor

The copy constructor is called, memory for the new array is allocated and stored in **x.p,** and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num**
sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location.) Remember that the copy constructor is called only for initializations.

For example,
this sequence does not call the copy constructor defined in the preceding program:

```
array a(10);
// ...
array b(10);
b = a; // does not call copy constructor
```

Question:Describe how Finding the address of an overloaded functions?Describe with example .
<u>Finding the Address of an Overloaded Function:</u>

If **myfunc()** is not overloaded, there is one and only one function called **myfunc(),** and the compiler has no difficulty assigning its address to **p**. However, if **myfunc()** is overloaded, how does the compiler know which version's address to assign to **p**? The answer is that it depends upon how **p** is declared. For example, consider this program:

```
#include <iostream>
using namespace std;
int myfunc(int a);
int myfunc(int a, int b);
int main()
{
int (*fp)(int a); // pointer to int f(int)
fp = myfunc; // points to myfunc(int)
cout << fp(5);
return 0;
}
int myfunc(int a)
{
return a;
}
int myfunc(int a, int b)
{
return a*b;
}
```

Here, there are two versions of **myfunc()** . Both return **int**, but one takes a single integer argument; the other requires two integer arguments. In the program, **fp** is declared as a pointer to a function that returns an integer and that takes one integer argument. When **fp** is assigned the address of **myfunc()** , C++ uses this information to select the **myfunc(int a)** version of **myfunc().** Had **fp** been declared like this:

int (*fp)(int a, int b);

Question:What is default argument? Ex-2011 or

Question: Write short not on default argument with example?

<u>Default argument:</u>

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value.

Following is a simple C++ example to demonstrate use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

```
#include<iostream>
using namespace std;
// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
   return (x + y + z + w);
}

/* Drier program to test above function*/
int main()
{
   cout << sum(10, 15) << endl;
   cout << sum(10, 15, 25) << endl;
   cout << sum(10, 15, 25, 30) << endl;
   return 0;
}
```

Output:

```
25
50
80
```

Once default value is used for an argument, all subsequent arguments must have default value.

Question: What are the different between Default arguments and Function overloading?

<u>Default arguments and overloading:</u>

A function that uses default parameters can count as a function with different numbers of parameters.

<u>Recall the three functions in the overloading example:</u>

```
int Process(double num);        // function 1
int Process(char letter);       // function 2
```

int Process(double num, int position); // function 3

Now suppose we declare the following function with default argument:
int Process(double x, int position= 5);   // function 4

This **function conflicts with function 3**, obviously. It ALSO **conflicts with function 1**. Consider these calls:

```
cout << Process(12.3, 10);          // matches functions 3 and 4
cout << Process(13.5);     // matches functions 1 and 4
```

So, function 4 cannot exist along with function 1 or function 3

Question: Describe the Ambiguity with Function overloading with example.
Function Overloading and Ambiguity:

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be ambiguous. Ambiguous statements are errors, and programs containing ambiguity will not compile.

By far the main cause of ambiguity involves C++'s automatic type conversions. Asyou know, C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function. For example, consider thisfragment:

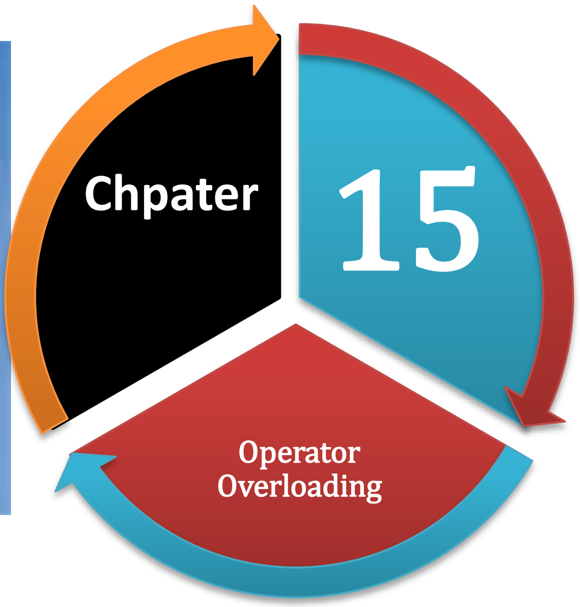
```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

As the comment indicates, this is not an error because C++ automatically converts the character **c** into its **double** equivalent. In C++, very few type conversions of this sort are actually disallowed. Although automatic type conversions are convenient, they are also a prime cause of ambiguity. For example, consider the following program:

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);
int main()
{
cout << myfunc(10.1) << " "; // unambiguous, calls
myfunc(double)
cout << myfunc(10); // ambiguous
return 0;
}
float myfunc(float i)
{
return i;
}
double myfunc(double i)
{
```

```
return -i;
}
```

Here, **myfunc()** is overloaded so that it can take arguments of either type **float** or type **double**. In the unambiguous line, **myfunc(double)** is called because, unless explicitly specified as **float**, all floating-point constants in C++ are automatically of type **double.**



Important Questions

1. What do you mean by operator overloading? What are the restrictions applied to operator overloading ? **Ex-2012**
2. What is operator overloading ? Why is it necessary? **Ex-2011**
3. Why overloading sometimes caused ambiguity? Which type of ambiguity may arise? **Ex-2011**
4. What are the advantages of overloading an operator using friend function? **Ex-2012**
5. What is the difference between early binding and late binding? **Ex-2011**
6. what are the differences between member operator function and friend operator functions? **Ex-2014**
7. What are 'unary' and 'binary' operator overloading? **Ex-2013**
8. Why ,if friend function is not used , then left-side argument of binary operator can neither be built-in data type nor be an object of the class other than class having the respective overloaded operator function. **Ex-2013**
9. Give an example of 'unary' operator overloading. **Ex-2013**

Question:What do you mean by operator overloading? What are the restrictions applied to operator overloading ? Ex-2012 or
Question: How can you create a member operator function?
<u>Operator Overloading:</u>

C++ allows you to specify more than one definition for an **operator** in the same scope, which is called **operator overloading** .

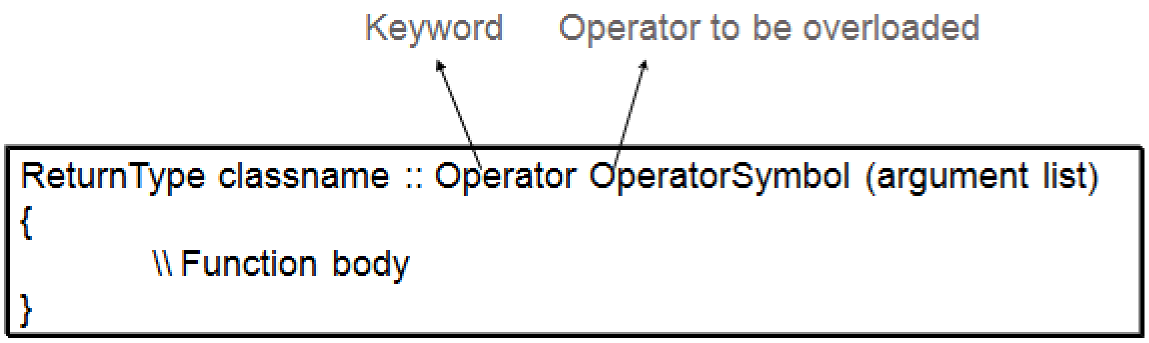
When you call an overloaded or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the operator with the parameter types specified in the definitions. The process of selecting the most appropriate operator is called overload resolution.

An operator function is created using the keyword operator. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class

<u>Creating a Member Operator Function:</u>

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
// operations
}
```


Keyword
Operator to be overloaded
ReturnType classname :: Operator OperatorSymbol (argument list)
{
\\ Function body
}

Operator Overloading Often, operator functions return an object of the class they operate on, but ret-type can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #.

For example, if you are overloading the / operator, use operator/.

When you are overloading a unary operator, arg-list will be empty.
When you are overloading binary operators, arg-list will contain one parameter.

First example of operator overloading.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
```

```
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

operator+() has only one parameter even though it overloads the binary + operator.The reason that operator+() takes only one parameter is that the operand on the left side of the + is passed implicitly to the function throughthe this pointer.
The operand on the right is passed in the parameter op2.The fact that the left operand is passed using this also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function. it is common for an overloaded operator function to return an object of the class it operates upon.
(ob1+ob2).show();

displays outcome of ob1+ob2 In this situation, ob1+ob2 generates a temporary object that ceases to exist after the call to show() terminates.

Restriction of Operator overloading:

- Not all operators can be overloaded. Only specific set of operators can be overloaded.
- Precedence of the operator cannot be changed on operator overloading.
- Operator Associativity cannot be changed on operator overloading.
- Number of arguments of an operator cannot be changed on operator overloading. For example –operator cannot be overloaded to accept only one operand.
- Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.

- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Question:What is operator overloading ? Why is it necessary? Ex-2011
Necessary of Operator Overloading:

Operator overloading reduced the learning curve and the defect rate , you can do cool stuff without writing much code.

Other benefit of operator overloading is that it allows us to seamlessly integrate a new class type into our programming environment. This type extensibility is an important part of the power of an oops languages.One operator is defined for a class, we can operate an object of that class using the normal C++ expression syntax. We can even use an object in expression involving other type of data.

Also You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive.
For example,
You can replace the code like:
calculation = add(multiply(a, b),divide(a, b));
to
calculation = (a*b)+(a/b);

Question: How Creating Prefix and Postfix Forms of the Increment and Decrement Operators Describe with Example?

Creating Prefix and Postfix Forms of the Increment and Decrement Operators:

Standard C++ allows you to explicitly create separate prefix and postfix versions of increment or decrement operators. To accomplish this, you must define two versions of the **operator++()** function. One is defined as shown in the foregoing program. The other is declared like this:
loc operator++(int x); If the **++** precedes its operand, the **operator++()** function is called. If the **++** follows its operand, the **operator++(int x)** is called and **x** has the value zero. The preceding example can be generalized. Here are the general forms for the prefix and postfix **++** and **– –** operator functions

Here are the general forms for the prefix and postfix **++** and **– –** operator functions.

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
// Postfix decrement
```

```
type operator– –(int x) {
// body of postfix operator
}
```

Question: Can You Overload operator using Shorthand ? how? Give a example.
Answer:
You can overload any of C++'s "shorthand" operators, such as **+=, –=,** and the like.
For example, this function overloads **+=** relative to **loc**:

```
loc loc::operator+=(loc op2)
{
longitude = op2.longitude + longitude;
latitude = op2.latitude + latitude;
return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Question:Why overloading sometimes caused ambiguity? Which type of ambiguity may arise? Ex-2011

Question:Give an example of 'unary' operator overloading. Ex-2013
Answer:

Here a program to find the complex numbers using unary operator overloading.

```
#include<iostream>

class complex
{
    int a,b,c;
   public:
      complex(){}
      void getvalue()
     {
             cout<<"Enter the Two Numbers:";
             cin>>a>>b;
     }



  void operator++()
     {
             a=++a;
             b=++b;
     }

     void operator--()
     {
```

```
                a=--a;
                b=--b;
            }

            void display()
            {
                cout<<a<<"+\t"<<b<<"i"<<endl;
             }
        };

        void main()
        {
            complex obj;
            obj.getvalue();
            obj++;
            cout<<"Increment Complex Number\n";
            obj.display();
            obj--;
            cout<<"Decrement Complex Number\n";
            obj.display();
        }
```

Output:

```
Enter the two numbers: 3 6
Increment Complex Number
4 +         7i
Decrement Complex Number
3 +         6i
```

Question:Is it Possible to Operator overloading using Friend function? How ? give a example.

<u>Operator Overloading Using a Friend Function:</u>

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

In this program, the **operator+()** function is made into a friend:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
```

```
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload assignment for

loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
```

```
return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the **=, ( ), [ ],** or **–>** operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Question: Explain with example how Unary operator overload using friend function?

Using a Friend to Overload ++ or – –:

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. his is because friend functions do not have **this** pointers. Assuming that you stay true to the original meaning of the **++** and – – operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed a **this** pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call. For example, this program uses friend functions to overload the prefix versions of **++** and **– –** operators relative to the **loc** class:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
```

```
loc operator++(loc &op)
{
op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```

Question:What are the advantages of overloading an operator using friend function? Ex-2012

Advantages of overloading an operator using friend function:

Question:what are the differences between member operator function and friend operator functions? Ex-2014

There are many difference between member operator function and friend operator functions.

Member operator function	Friend operator function

Question:What are 'unary' and 'binary' operator overloading? Ex-2013

Unary operator:

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.

- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as **in !obj, -obj, and ++obj** but sometime they can be used as postfix as well like **obj++ or obj--.**
Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
using namespace std;

class Distance {
  private:
    int feet;           // 0 to infinite
    int inches;         // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();    // display D1

  -D2;                  // apply negation
  D2.displayDistance();    // display D2

  return 0;
}
```

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

```
F: -11 I:-10
```

```
F: 5 I:-11
```

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

<u>Binary Operator:</u>

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently **like addition (+)** operator, **subtraction (-) operator** and **division (/)** operator.

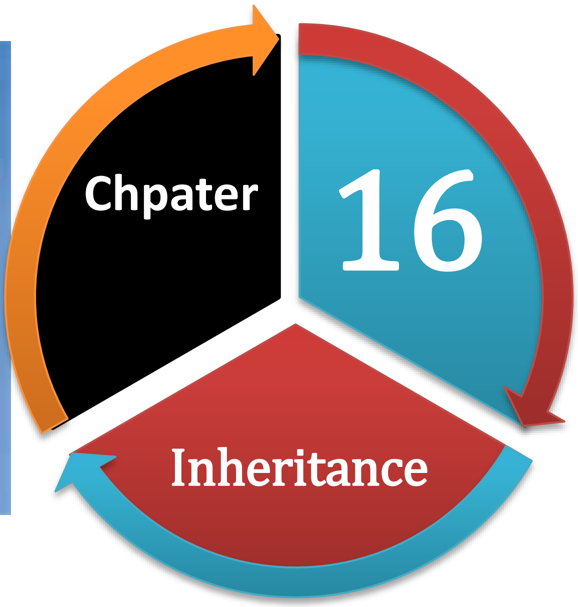
Following example explains how **addition (+) operator** can be overloaded. Similar way, you can overload **subtraction (-) and division (/) operators**.

```
#include <iostream>
using namespace std;

class Box {
  double length;      // Length of a box
  double breadth;     // Breadth of a box
  double height;      // Height of a box

public:

  double getVolume(void) {
    return length * breadth * height;
  }

  void setLength( double len ) {
    length = len;
  }

  void setBreadth( double bre ) {
    breadth = bre;
  }

  void setHeight( double hei ) {
    height = hei;
  }

  // Overload + operator to add two Box objects.
  Box operator+(const Box& b) {
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
  }
};

// Main function for the program
int main( ) {
  Box Box1;             // Declare Box1 of type Box
  Box Box2;            // Declare Box2 of type Box
  Box Box3;            // Declare Box3 of type Box
  double volume = 0.0;    // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Question:Why ,if friend function is not used , then left-side argument of binary operator can neither be built-in data type nor be an object of the class other than class having the respective overloaded operator function. Ex-2013



Important Questions

1. Is there any way to access private member of a class without taking help of own member function of that class ? Explain your answer with an example.. **Ex-2011**
2. What are multiple , multilevel and hybrid inheritance? **Ex-2011**
3. Explain how 'virtual inheritance ' can solve the problem that is caused when any member of base class may be inherited in different ways to a 'high level derived class'. **Ex-2011**
4. Explain why protected access specifier is used while inheriting a base class? **Ex-2012**
5. Can pointer to a base class be assigned to a pointer to derived class, or, vice versa? **Ex-2013**
6. If both the base class and derived class have a member of same name , which member of the respective class can be and can't be , accessible by the above pointer assignment . **Ex-2013**
7. In derived class constructors and distractors are called in which order? **Ex-2012**
8. Explain the accessing mechanism of data members and member functions in case of Inside main() function and Inside a member function of the same class. **Ex-2013**
9. What is multiple inheritance (virtual inheritance) ? What are its advantages and disadvantages ? **Ex-2012**
10. Briefly discuss inheritance with an example. **Ex-2013**
11. Explain why protected access specifier is used while inheriting a base class? **Ex-2012**
12. In derived class constructors and distractors are called in which order? **Ex-2012**
13. Why is destructor function executed in reverse order of derivation? **Ex-2013**
14. What are the two ways in which a derived class can inherit more than one base class? **Ex-2013**
15. Is there any way to access private member of a class without taking help of own member function of that class ? Explain your answer with an example.. **Ex-2011**
16. What are multiple , multilevel and hybrid inheritance? **Ex-2011**

Question: What is inheritance?Discuss with Example?
Question:Briefly discuss inheritance with an example. Ex-2013

Inheritence:

Inheritance in Object Oriented Programming can be described as a process of creating new classes from existing classes. New classes inherit some of the properties and behavior of the existing classes. An existing class that is "parent" of a new class is called a base class. New class that inherits properties of the base class is called a derived class.
or

Derive quality and characteristics from parents or ancestors. Like you inherit features of your parents.
Example: "She had inherited the beauty of her mother"

Inheritance is a technique of code reuse. It also provides possibility to extend existing classes by creating derived classes.

Inheritance Syntax:
The basic syntax of inheritance is:

class DerivedClass : accessSpecifier BaseClass

Access specifier can be **public, protected and private**. The default access specifier is **private.** Access specifiers affect accessibility of data members of base class from the derived class. In addition, it determines the accessibility of data members of base class outside the derived class.

Question: Describe the Inheritence Access Specifier? Or
Question:Explain why protected access specifier is used while inheriting a base class? Ex-2012
Inheritance Access Specifiers:
Public Inheritance:
This inheritance mode is used mostly. In this the protected member of Base class becomes protected members of Derived class and public becomes public.

class DerivedClass : public BaseClass

Accessing Base class members	public	protected	private
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Public)	Yes (As Protected)	No
From object of a Derived class	Yes	No	No
From Derived class of Derived Classes	Yes (As Public)	Yes (As Protected)	No

Derived class of Derived Classes: If we are inheriting a derived class using a public inheritance as shown below

class B : public A
class C : public B

then public and protected members of class A will be accessible in class C as public and protected respectively.

Protected Inheritance:

In protected mode, the public and protected members of Base class becomes protected members of Derived class.

class DerivedClass : **protected** BaseClass

Accessing Base class members	**public**	**protected**	**private**
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Protected)	Yes (As Protected)	No
From object of a Derived class	No	No	No
From Derived class of Derived Classes	Yes (As Protected)	Yes (As Protected)	No

Derived class of Derived Classes: If we are inheriting a derived class using a public inheritance as shown below

class B : protected A
class C : protected B

then public and protected members of class A will be accessible in class C as protected

<u>**Private Inheritance:**</u>
In private mode the public and protected members of Base class become private members of Derived class.

class DerivedClass : **private** BaseClass
class DerivedClass : BaseClass // By default inheritance is private

Accessing Base class members	**public**	**protected**	**private**
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Private)	Yes (As Private)	No
From object of a Derived class	No	No	No
From Derived class of Derived Classes	No	No	No

Derived class of Derived Classes: If we are inheriting a derived class using a public inheritance as shown below
class B : private A
class C : private B
then public and protected members of class A will not be accessible in class C

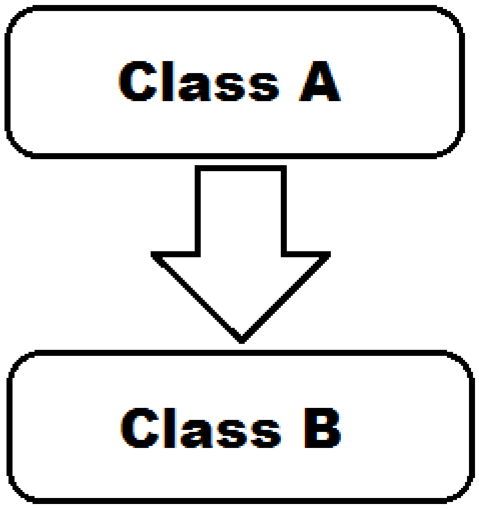
Question:Define different type of inheritance with example?

<u>**Types of Inheritance:**</u>
There are different types of inheritance:

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid (Virtual) Inheritance**

<u>**1.Single Inheritance:**</u>

Single inheritance represents a form of inheritance when there is only one base class and one derived class. For example, a class describes a **Person:**

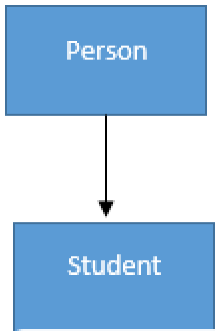


Example of Single Inheritance:

```
//base class
class Person
{
public:
        Person(string szName, int iYear)
        {
                m_szLastName = szName;
                m_iYearOfBirth = iYear;
        }
        string m_szLastName;
        int m_iYearOfBirth;
        void print()
        {
                cout << "Last name: " << szLastName << endl;
                cout << "Year of birth: " << iYearOfBirth << endl;
        }
protected:
        string m_szPhoneNumber;
};
```

We want to create new class Student which should have the same information as Person class plus one new information about university. In this case, we can create a derived class Student:

```
//derived class
class Student:public Person
{
public:
	string m_szUniversity;
};
```



Class Student is having access to all the data members of the base class (Person). Since class Student does not have a constructor so you can create a constructor as below

```
//will call default constructor of base class automatically
Student(string szName, int iYear, string szUniversity)
{
	m_szUniversity = szUniversity;
}
```

If you want to call the parameterized(user defined) constructor of a base class from a derived class then you need to write a parameterized constructor of a derived class as below

```
Student(string szName, int iYear, string szUniversity)
:Person(szName, iYear)
{
	m_szUniversity = szUniversity;
}
```

Person(szName, iYear) represents call of a constructor of the base class **Person**. The passing of values to the constructor of a base class is done via member initialization list.

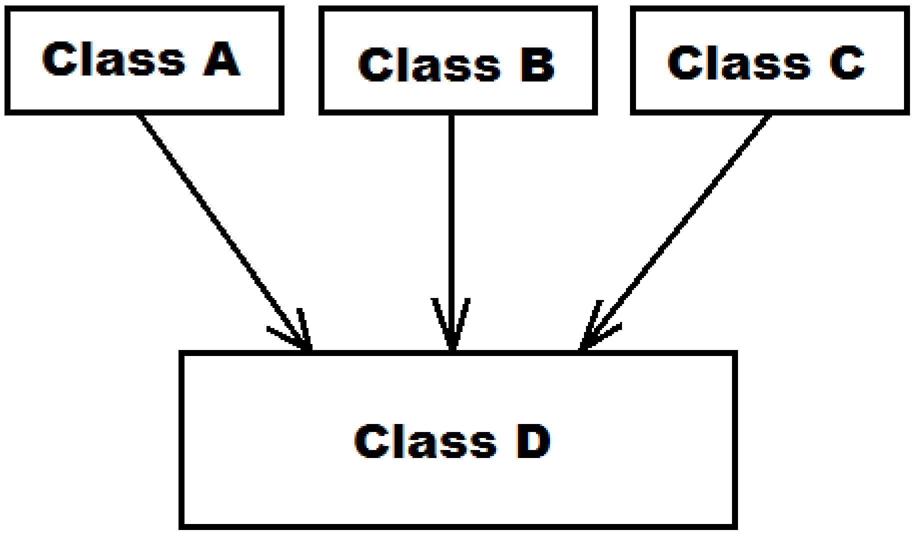
We can access member functions of a base class from a derived class. For example, we can create a new **print()** function in a derived class, that uses **print()** member function of a base class:

```
void print()
{
	//call function print from base class
	Person::print();
	cout << "University " << m_szUniversity << endl;
}
```

If you want to call the member function of the base class then you have to use the name of a base class

Multiple Inheritance:

Multiple inheritance represents a kind of inheritance when a derived class inherits properties of **multiple** classes. For example, there are three classes A, B and C and derived class is D as shown below:



If you want to create a class with multiple base classes, you have to use following syntax:

Class DerivedClass: accessSpecifier BaseClass1, BaseClass2, …, BaseClassN

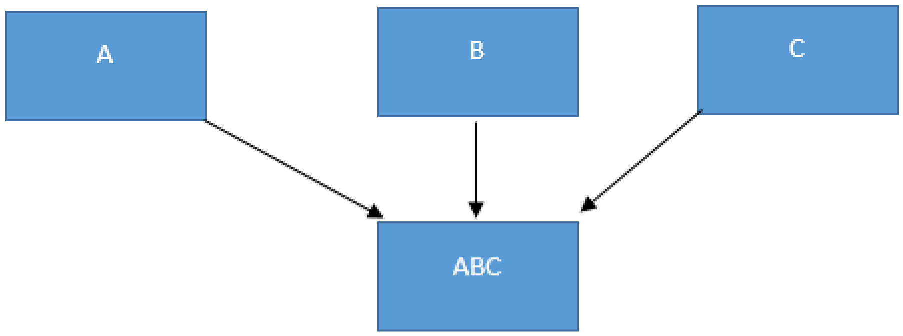
Example of Multiple Inheritance

```
class A
{
        int m_iA;
        A(int iA) :m_iA(iA)
        {
        }
};

class B
{
        int m_iB;
        B(int iB) :m_iB(iB)
        {
        }
};

class C
{
        int m_iC;
        C(int iC) :m_iC(iC)
```

```
                {
                }
        };
```

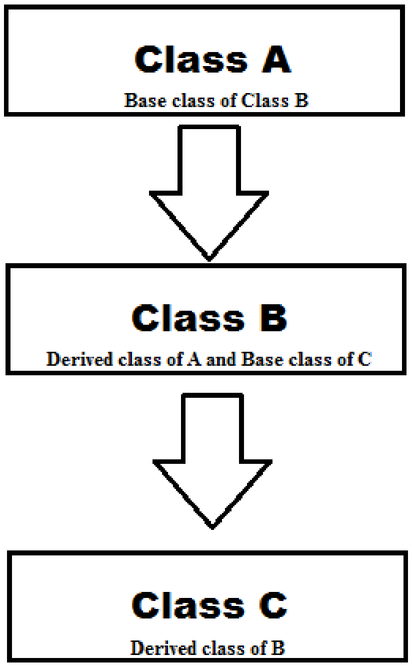
You can create a new class that will inherit all the properties of all these classes:

```
        class ABC :public A, public B, public C
        {
                int m_iABC;
                //here you can access m_iA, m_iB, m_iC
        }
```



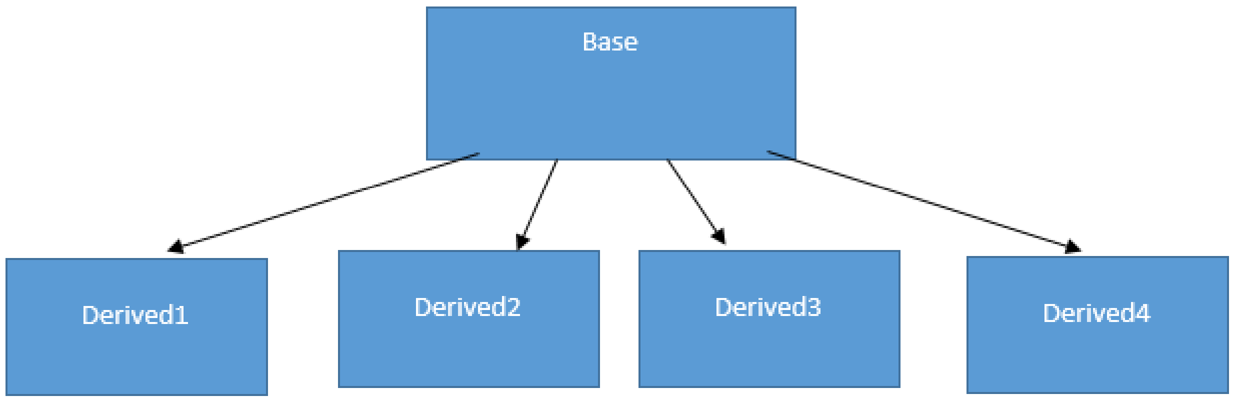
Multilevel Inheritance:

Multilevel inheritance represents a type of inheritance when a Derived class is a base class for another class. In other words, deriving a class from a derived class is known as multi-level inheritance. Simple multi-level inheritance is shown in below image where Class A is a parent of Class B and Class B is a parent of Class C



Example of Multi-Level Inheritance

Below Image shows the example of multilevel inheritance



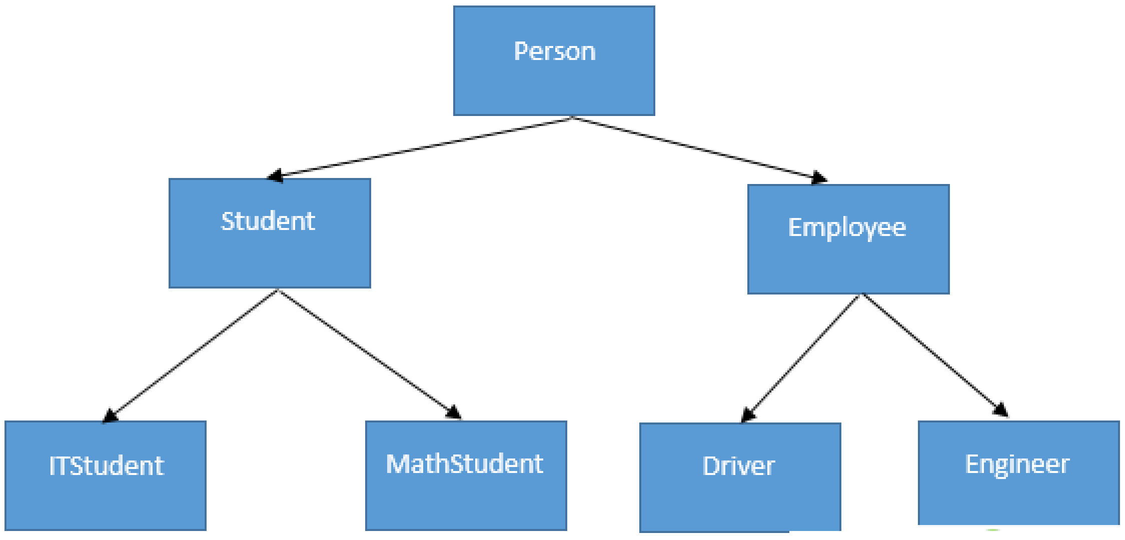
As you can see, Class **Person** is the base class of both **Student** and **Employee** classes. At the same time, Class **Student** is the base class for **ITStudent** and **MathStudent** classes. **Employee** is the base class for **Driver** and **Engineer** classes.
The code for above example of multilevel inheritance will be as shown below

```
class Person


{
        //content of class person
};

class Student :public Person
{
        //content of Student class
};

class Employee : public Person
{
        //content of Employee class
};

class ITStundet :public Student
{
        //content of ITStudent class
};

class MathStundet :public Student
```

```
{
        //content of MathStudent class
};

class Driver :public Employee
{
        //content of class Driver
};

class Engineer :public Employee
{
        //content of class Engineer
};
```

Hierarchical Inheritance:

When there is a need to create multiple Derived classes that inherit properties of the same Base class is known as Hierarchical inheritance

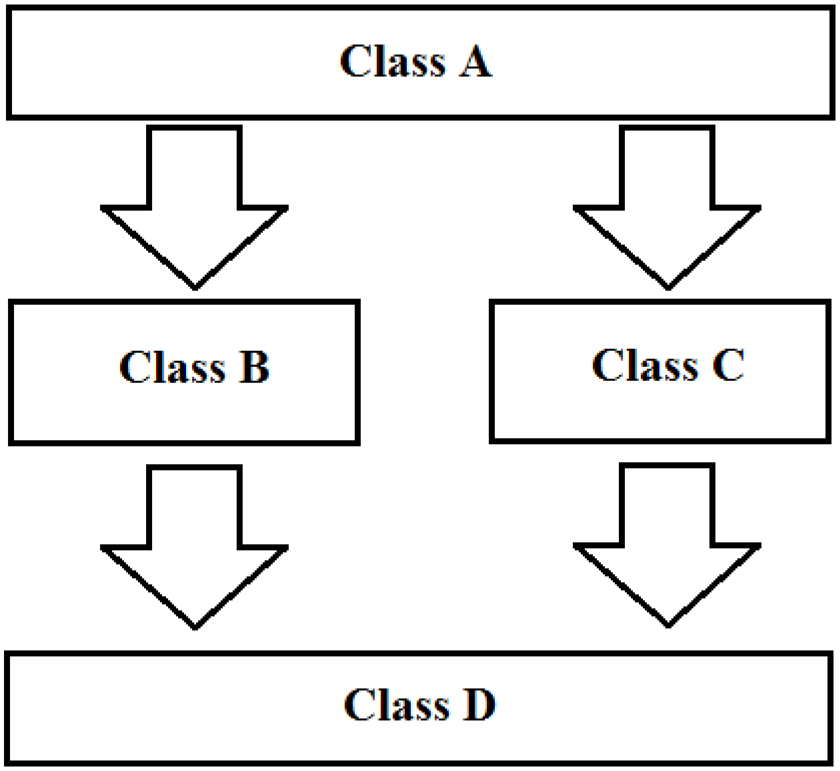


```
class base
{
        //content of base class
};

class derived1 :public base
{
        //content of derived1
};

class derived2 :public base
{
        //content of derived
};
```

```
class derived3 :public base
{
        //content of derived3
};

class derived4 :public base
{
        //content of derived4
};
```

Hybrid Inheritance (also known as Virtual Inheritance):

Combination of Multi-level and Hierarchical inheritance will give you Hybrid inheritance



Base-Class Access Control:

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name {
// body of class
};
```

The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class. For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**:

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()

{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class. For example, the following program will not even compile because both **set()** and **show()**
are now private elements of **derived**:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";}
};
// Public elements of base are private in derived.
class derived : private base {
int k;

public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // error, can't access set()
ob.show(); // error, can't access show()
return 0;
}
```

When a base class' access specifier is **private,** public and protected members of thebase become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.

Question:Explain what happened if protected access specifier is used while inheriting a base class? Ex-

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class.

```
#include <iostream>
using namespace std;
class base {
Remember
protected:
int i, j; // private to base, but accessible by derived
public:
```

```
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected, derived**'s function **setk()** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile. When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j.**

```
 #include <iostream>
using namespace std;
class base {
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base {
int k;
public:
void setk() { k = i*j; } // legal
void showk() { cout << k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
int m;
public:
void setm() { m = i-j; } // legal
void showm() { cout << m << "\n"; }
};
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(2, 3);
ob1.show();
ob1.setk();
ob1.showk();
ob2.set(3, 4);
ob2.show();
ob2.setk();
ob2.setm();
ob2.showk();
ob2.showm();
return 0;
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2.** (However, **i** and **j** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// Now, all elements of base are private in derived1.
class derived1 : private base {
int k;
public:
// this is legal because i and j are private to derived1
void setk() { k = i*j; } // OK
void showk() { cout << k << "\n"; }
};
// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
int m;
public:
// illegal because i and j are private to derived1
void setm() { m = i-j; } // Error
void showm() { cout << m << "\n"; }
};
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(1, 2); // error, can't use set()
ob1.show(); // error, can't use show()
ob2.set(3, 4); // error, can't use set()
ob2.show(); // error, can't use show()
return 0;
}
```

Even though **base** is inherited as **private** by **derived1, derived1** still has access to **base**'s **public** and **protected** elements. However, it cannot pass along this privilege.

Question:Explain why protected access specifier is used while inheriting a base class? Ex-2012
Answer:
When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class.

```
#include <iostream>
using namespace std;
class base {
Remember
protected:
int i, j; // private to base, but accessible by derived
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected, derived**'s function **setk()** may access them.

Question: Is it possible passing parameters to Base-Class Control? Explaine with Example.
Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructor functions that require arguments. In cases where only the derived class' constructor requires one or more parameters,

The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
        {
// body of derived constructor
}
```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class.

Consider this program:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived()** uses only **x; y** is passed along to **base().**

Any parameters required by the base class are passed to it in the base class' argument list specified after the colon.
Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
```

```
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived class are simply passed along to the base. For example, in this program, the derived class' constructor
takes no arguments, but **base1()** and **base2()** do:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
/* Derived constructor uses no parameter,
but still must be declared as taking them to
pass them along to base classes.
*/
derived(int x, int y): base1(x), base2(y)
{ cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

A derived class' constructor function is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
int j;
public:
// derived uses both x and y and then passes them to base.
derived(int x, int y): base(x, y)
{ j = x*y; cout << "Constructing derived\n"; }
```

Question:What do you mean by Granting Access? Explain with example.
Or
Question:Is it possible to private member variable use from main function? If possible how? Explain with example
<u>Granting Access:</u>

When a base class is inherited as **private**, all public and protected members of that class ecome private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification.

For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**.

Two ways to accomplish this.:
First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces .
The second way to restore an inherited member's **access specification** is to employ an access declaration within the derived class.

An access declaration takes this general form:

base-class::member

The access declaration is put under the appropriate access heading in the derived class' declaration.
To see how an access declaration works, let's begin with this short fragment:

```
class base {
public:
int j; // public in base
};
// Inherit base as private.
class derived: private base {
public:
// here is access declaration
base::j; // make j public again
.
.
.
};
```

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

base::j;

Question:What are the two ways in which a derived class can inherit more than one base class? Ex-2013
Answer:

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{

derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
// i ambiguous here, too
ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base.** However, **derived3** inherits both **derived1** and **derived2.** This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like ob.i = 10;

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program:

THE FIRST IS TO APPLY THE SCOPE RESOLUTION:

The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.derived1::i = 10; // scope resolved, use derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, because the **::** was applied, the program has manually selected **derived1**'s version of **base.**

2ND IS USING VIRTUAL CLASS:

_Virtual Base Classes

```
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

Question:Explain how 'virtual inheritance ' can solve the problem that is caused when any member of base class may be inherited in different ways to a 'high level derived class'. Ex-2011 or

Question:What is multiple inheritance (virtual inheritance) ? What are its advantages and disadvantages ? Ex-2012

2ND IS USING VIRTUAL CLASS:

_Virtual Base Classes

```
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

Protected Base-Class Inheritance

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class.

For example,

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
```

```
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base{
int k;
public:
// derived may access base's i and j and setij().
void setk() { setij(10, 12); k = i*j; }
// may access showij() here
void showall() { cout << k << " "; showij(); }
};
int main()
ote
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
// member of derived
return 0;
}
```

As you can see by reading the comments, even though **setij()** and **showij()** are public members of **base,** they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main()** .

Inheriting Multiple Base Classes:

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2.**

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};

class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};

// Inherit multiple base classes.
Class derived: public base1, public base2 {
public:
void set(int I, int j) { x=I; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

Constructors, Destructors, and Inheritance:

There are two major questions that arise relative to constructors and destructors when inheritance is involved.
First, when are base-class and derived-class constructor and destructor functions called?
Second, how can parameters be passed to base-class constructor functions?
This section examines these two important topics.

Question:In derived class constructors and distractors are called in which order? Ex-2012
or
Question:When Constructor and Destructor Functions Are Executed?
<u>Answer:</u>
It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions.

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

As the comment in **main()** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

Question:Why is destructor function executed in reverse order of derivation? Ex-2013

Passing Parameters to Base-Class Constructors:

how do you pass arguments to a constructor in a base class?
The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}
```

base1 through baseN are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

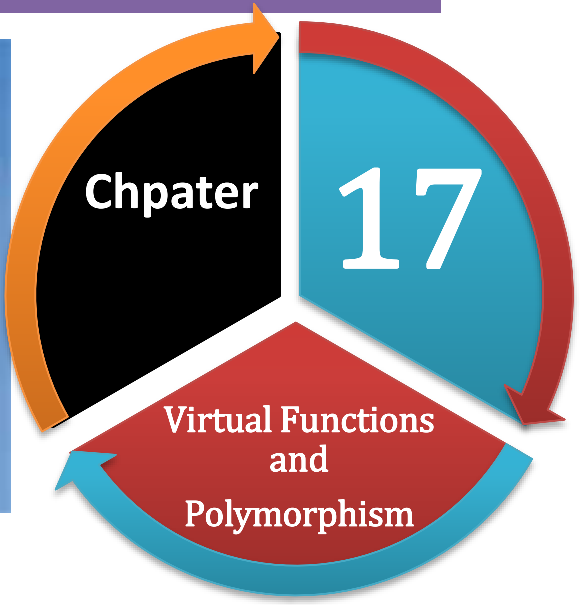
```
#include <iostream>
    using namespace std;
    class base {
    protected:
    int i;
    public:
    base(int x) { i=x; cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }
    };
    class derived: public base {
    int j;
    public:
    // derived uses x; y is passed along to base.
    derived(int x, int y): base(y)
    { j=x; cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
    void show() { cout << i << " " << j << "\n"; }
    };
    int main()
    {
    derived ob(3, 4);
    ob.show(); // displays 4 3
    return 0;
    }
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**.However, **derived()** uses only **x; y** is passed along to **base().** In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class

requires it. In this situation, the arguments passed to the derived class are simply passed along to the base.



Important Questions

1. Explain ,with an example ,what is the rule of 'virtual function' for above situation. **Ex-2013**
2. What is a "virtual member function"? **Ex-2012**
3. What is pure virtual function? What is abstract class? **Ex-2012**
4. What is the difference between virtual function and overloading function? **Ex-2012**
5. What is a pure virtual function ? How a pure virtual function is declared? **Ex-2012**
6. Distinguish between runtime and compile time polymorphism ? How runtime polymorphism is achieved? **Ex-2012**
7. Define early binding and late binding example? **Ex-2012**
8. **Question:What is the difference between early binding and late binding? Ex-2011**
9. Explain different types of polumorphism in C++. **Ex-2013**
10. bCreate an abstract base class shape with two members base and height,a member function for initializing and a pure virtual function to compute area().Derive two specific xlasses Triangle and Rectangle Which override the function area().Use these classes in a main function and display the area of a Triangle and a Rectangle. **Ex-2013**
11. Distinguish between static binding and dynamic binding. **Ex-2013**

Question:What is a "virtual member function"? Ex-2012 or
Question:What is a Virtual Function explain with example?

Virtual function:

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer.

When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at run
time.

Example:

```
 include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
  p = &d1
vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

inside **base**, the virtual function **vfunc()** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When
vfunc() is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed.

Question: How virtual Function Calling through a Base Class Reference Describe with Example?
<u>Calling a Virtual Function Through a Base Class Reference:</u>
Virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference.

The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter. For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
a virtual function. */
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
```

```
cout << "This is derived2's vfunc().\n";
}
};
// Use a base class reference parameter.
void f(base &r) {
r.vfunc();
}
int main()
{
base b;
derived1 d1;
derived2 d2;
f(b); // pass a base object to f()
f(d1); // pass a derived1 object to f()
f(d2); // pass a derived2 object to f()
return 0;
}
```

This program produces the same output as its preceding version. In this example, the function **f()** defines a reference parameter of type **base.** Inside **main()** , the function is called using objects of type **base**, **derived1**, and **derived2.** Inside **f()** , the specific version of **vfunc()** that is called is determined by the type of object being referenced when the function is called.

Question:u What do you mean by Virtual Attribute Is inherited How its work discuss with example?

<u>The Virtual Attribute Is Inherited:</u>

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual. For example, consider this program:

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
/* derived2 inherits virtual function vfunc()
from derived1. */
class derived2 : public derived1 {
public:
```

```
// vfunc() is still virtual
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc()** is still virtual.

Virtual Functions Are Hierarchical:

when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which **derived2** does not override **vfunc()** :

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
```

```
public:
// vfunc() not overridden by derived2, base's is used
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use base's vfunc()
return 0;
}
```

The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because **derived2** does not override **vfunc()** , the function defined by **base** is used when **vfunc()** is referenced relative to objects of type **derived2**

Question:What is a pure virtual function ? How a pure virtual function is declared? Ex-2012
Question: What is pure virtual function? What is abstract class? Ex-2012

Pure Virtual Function:

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

"A pure virtual function is a virtual function that has no definition within the base class". To declare a pure virtual function, use this general form:

virtual type func-name(parameter-list) = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function. The base class, **number**, contains an integer called **val**, the function **setval()** , and the **pure virtual function show()** . The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show()** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
using namespace std;
class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
```

```
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 - hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, number simply provides the common interface for the derived types to use. There is no reason to define show() inside number since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making show() pure also ensures that all derived classes will indeed redefine it to meet their own needs. Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

Abstract Classes:

A class that contains at least one pure virtual function is said to be abstract. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Question: what do you mean by "one interface, multiple methods" discuss with appropriate example?

Using Virtual Functions:

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type. **One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism.** Using these features, you create a class hierarchy that moves from general to specific (base to derived).

Example:

Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details. Following is a simple example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class **convert** declares two variables, **val1** and **val2**, which hold the initial and converted values, respectively. It also defines the functions **getinit()** and

getconv() , which return the initial value and the converted value. These elements of **convert** are fixed and applicable to all derived classes that will inherit **convert.** However, the function that will actually perform the conversion, **compute()** , is a pure virtual function that must be defined by the classes derived from **convert.** The specific nature of **compute()** will be determined by what type of conversion is taking place.

Virtual function practical example.

```
#include <iostream>
using namespace std;
class convert {
protected:
double val1; // initial value
double val2; // converted value
public:
convert(double i) {
val1 = i;
}
double getconv() { return val2; }
double getinit() { return val1; }
virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
public:
l_to_g(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.7854;
}
};
// Fahrenheit to Celsius
class f_to_c : public convert {
458 C + + : T h e C o m p l e t e R e f e r e n c e
public:
f_to_c(double i) : convert(i) { }
void compute() {
val2 = (val1-32) / 1.8;
}
};
int main()
{
convert *p; // pointer to base class
l_to_g lgob(4);
f_to_c fcob(70);
// use virtual function mechanism to convert
p = &lgob;
cout << p->getinit() << " liters is ";
p->compute();
cout << p->getconv() << " gallons\n"; // l_to_g
p = &fcob;
cout << p->getinit() << " in Fahrenheit is ";
p->compute();
cout << p->getconv() << " Celsius\n"; // f_to_c
return 0;
}
```

The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute()** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant. One of the benefits of derived classes and virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
f_to_m(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.28;
}
};
```

An important use of abstract classes and virtual functions is in *class libraries*. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs. One final point: The base class **convert** is an example of an abstract class. The virtual function **compute()** is not defined within **convert** because no meaningful definition can be provided. The class **convert** simply does not contain sufficient information for **compute()** to be defined. It is only when **convert** is inherited by a derived class that a complete type is created.

Question:Define early binding and late binding example? Ex-2012
Question:Question:What is the difference between early binding and late binding? Ex-2011
Question:Distinguish between static binding and dynamic binding. Ex-2013
Answer:

Early vs. Late Binding:
There are two terms that need to be defined because they are used frequently in discussions of C++ and object-oriented programming: early binding and late binding.

Early Binding:
Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.)
Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

late binding. Or Dynamic Binding:

The opposite of early binding is late **binding.** As it relates to C++, **late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding.** As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. **The main advantage of late binding is flexibility.** Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

Question:What is the difference between virtual function and overloading function? Ex-2012
Question:Distinguish between runtime and compile time polymorphism ? How runtime polymorphism is achieved? Ex-2012
Question:Explain different types of polymorphism in C++. Ex-2013
Question:bCreate an abstract base class shape with two members base and height,a member function for initializing and a pure virtual function to compute area().Derive two specific xlasses Triangle and Rectangle Which override the function area().Use these classes in a main function and display the area of a Triangle and a Rectangle. Ex-2013

Important Questions

1. What is 'function template' ? **Ex-2013**
2. Give an example of 'function template'. **Ex-2013**
3. What's the idea behind templates ? What's the syntax /semantics for a "class template"? **Ex-2012**
4. Define a swap function template for swapping two objects of the same type. **Ex-2012**)How can you create a function template in C++? **Ex-2013**
5. Give general syntax of a generic class. **Ex-2013**

Question:What is 'function template' ? Ex-2013 or

Question:What's the idea behind templates ? What's the syntax /semantics for a "class template"? Ex-2012

Template:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept. There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as

Classes, let us see how they work:
Function Template
The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

Question:Give an example of 'function template'. Ex-2013
Question:How can you create a function template in C++? Ex-2013

Function Template:

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>
#include <string>
using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
return a < b ? b:a;
}


 int main (){
int i = 39;
int j = 20;
cout << "Max(i, j): " << Max(i, j) << endl;
double f1 = 13.5;
double f2 = 20.7;
cout << "Max(f1, f2): " << Max(f1, f2) << endl;
string s1 = "Hello";
string s2 = "World";
cout << "Max(s1, s2): " << Max(s1, s2) << endl;
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Question:What's the syntax /semantics for a "class template"? Ex-2012

Answer:

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name {
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma separated list.
Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
using namespace std;
template <class T>
class Stack {
private:
vector<T> elems; // elements
public:
void push(T const&); // push element
void pop(); // pop element
T top() const; // return top element
bool empty() const{ // return true if empty.
return elems.empty();
}
};
template <class T>
void Stack<T>::push (T const& elem)
{
// append copy of passed element
elems.push_back(elem);
}
template <class T>
void Stack<T>::pop ()
{
if (elems.empty()) {
throw out_of_range("Stack<>::pop(): empty stack");
}
// remove last element
elems.pop_back();
}
template <class T>
T Stack<T>::top () const
{
if (elems.empty()) {
throw out_of_range("Stack<>::top(): empty stack");
}
// return copy of last element
return elems.back();
}
int main()
{
try {
Stack<int> intStack; // stack of ints
Stack<string> stringStack; // stack of strings
// manipulate int stack
intStack.push(7);
cout << intStack.top() <<endl;
// manipulate string stack
stringStack.push("hello");
cout << stringStack.top() << std::endl;
stringStack.pop();
stringStack.pop();
}
catch (exception const& ex) {
cerr << "Exception: " << ex.what() <<endl;
return -1;
}
}
```

If we compile and run above code, this would produce the following result:

```
7
hello
Exception: Stack<>::pop(): empty stack
```

Question:Define a swap function template for swapping two objects of the same type. Ex-2012

Answer:

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the general process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j);          // swap integers
swapargs(x, y);          // swap floats
swapargs(a, b);          // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}
```

Let's look closely at this program
The line:

template <class X> void swapargs(X &a, X &b) tells the compiler two things:

- **That a template is being created and**
- **That a generic definition is beginning.**

Here, X is a generic type that is used as a placeholder. After the template portion, the function swapargs() is declared, using X as the data type of the values that will be swapped.

In main() , **the swapargs() function is called using three different types of data: ints, doubles, and chars**. Because **swapargs() is a generic function,** the compiler automatically creates three versions of swapargs() : one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.
Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a template statement) is also called a *template function*. Both terms will be used interchangeably. When the compiler creates a specific version of this function, it is said to have created a *specialization.* This is also called a *generated function*. The act of generating a function is referred to as *instantiating* it. Put differently, a generated function is a specific instance of a template function.

Question: What is Generic Function ?Write a syntax of Generic function.
<u>**Generic Functions:**</u>
A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter.

Through a generic function, a single general procedure can be applied to a wide range of data**. By creating a generic function, you can define the nature of the algorithm, independent of any data.** Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, **when you create a generic function you are creating a function that can automatically overload itself.**

A generic function is created using the keyword template. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.

The general form of a template function definition is shown here:

```
<class Ttype> ret-type func-name(parameter list)
{
// body of function
}
```

Here, ***Ttype*** **is a placeholder name for a data type used by the function.** This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function. Although the use of the keyword **class** to specify a generic type
in a **template** declaration is traditional, you may also use the keyword **typename**.

Question:Give general syntax of a generic class. Ex-2013
Generic Class:

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class.
When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created. The general form of a generic class declaration is shown here

```
template <class Ttype> class class-name {
.
..
}
```

Here, Ttype is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list. Once you have created a generic class, you create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Here, type is the type name of the data that the class will be operating upon. Member functions of a generic class are themselves automatically generic. You need not use **template** to explicitly specify them as such

Important Questions

1. What are some ways try / catch/ throw can improve software quality ? **Ex-2012**
2. How should I handle resources if my constructors may throw exceptions? **Ex-2012**
3. What is an exception?Why do we need to handle exception? **Ex-2013**
4. what are the keywords used in C++ for executing handling?Describe their usage with suitable example. **Ex-2013**

Explain how exception is handling by using 'try' , 'catch' and 'throw' ? **Ex-2013**

Question:What is an exception?Why do we need to handle exception? Ex-2013
Exception:

An exception is a problem that arises during the execution of a program. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

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

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

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

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

Example of Exception Handling:

```
#include <iostream>
using namespace std;

int main()
{
  int x = -1;

  // Some code
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Why need Exception Handling:

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)
3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Question:Describe Fundamental of Exception Handling? Or

Question:what are the keywords used in C++ for executing handling? Describe their usage with suitable example. Ex-2013 or

Exception Handling Fundamentals:

Exception handling is built upon three keywords: try, catch, and throw. In the most general terms, program statements that you want to monitor for exceptions are contained in a try block. If an exception (i.e., an error) occurs within the try block, it is thrown (using throw). The exception is caught, using catch, and processed.

The general form of try and catch are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}
```

```
..
.
catch (typeN arg) {
// catch block
}
```

Try:

The try can be as short as **a few statements within one function or as all encompassing as enclosing the main()** function code within a try block (which effectively causes the entire program to be monitored).

Catch:

When an exception is thrown, **it is caught by its corresponding catch statement**, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception. That is, if the data type specified by a catch matches that of the exception, then that catch statement is executed (and all others are bypassed).

Throw:

The general form of the throw statement is shown here:

throw exception;

throw generates the exception specified by exception. If this exception is to be caught, then throw must be executed either from within a try block itself, or from any function called from within the try block (directly or indirectly).

Here is a simple example that shows the way C++ exception handling operates.

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Question:Explain how exception is handling by using 'try' , 'catch' and 'throw' ? Ex-2013

Example:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Explain how exception is handling by using 'try' , 'catch' and 'throw'

Look carefully at this program:

As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is not called. Rather, program execution is transferred to it. (The program's stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.

Question:What are some ways try / catch/ throw can improve software quality ? Ex-2012

Answer:

The commonly used alternative to try / catch / throw is to return a *return code* (sometimes called an *error code*) that the caller explicitly tests via some conditional statement such as if. For example, printf(), scanf() and malloc() work this way: the caller is supposed to test the return value to see if the function succeeded.

Although the return code technique is sometimes the most appropriate error handling technique, there are some nasty side effects to adding unnecessary if statements:

- **Degrade quality:** It is well known that conditional statements are approximately ten times more likely to contain errors than any other kind of statement. So all other things being equal, if you can eliminate conditionals / conditional statements from your code, you will likely have more robust code.

- **Slow down time-to-market:** Since conditional statements are branch points which are related to the number of test cases that are needed for white-box testing, unnecessary conditional statements increase the amount of time that needs to be devoted to testing. Basically if you don't exercise every branch point, there will be instructions in your code that will *never* have been executed under test conditions until they are seen by your users/customers. That's bad.
- **Increase development cost:** Bug finding, bug fixing, and testing are all increased by unnecessary control flow complexity.

So compared to error reporting via return-codes and if, using try / catch / throw is likely to result in code that has fewer bugs, is less expensive to develop, and has faster time-to-market. Of course if your organization doesn't have any experiential knowledge of try / catch / throw, you might want to use it on a toy project first just to make sure you know what you're doing — you should always get used to a weapon on the firing range before you bring it to the front lines of a shooting war.

Question:How should I handle resources if my constructors may throw exceptions? Ex-2012

Answer:

a constructor throws an exception, the object's destructor is not run. If your object has already done something that needs to be undone (such as allocating some memory, opening a file, or locking a semaphore), this "stuff that needs to be undone" must be remembered by a data member inside the object.

For example, rather than allocating memory into a raw Fred* data member, put the allocated memory into a "smart pointer" member object, and the destructor of this smart pointer will delete the Fred object when the smart pointer dies. The By the way, if you think your Fred class is going to be allocated into a smart pointer, be nice to your users and create a typedef within your Fred class

```
class Foo
{
public:
   Foo()
   {
     int error = 0;
      p = new Fred;

      throw error;  // Force throw , trying to understand what will
happen to p
   }

  ~Foo()
   {
     if (p)
     {
        delete p;
        p = 0;
     }
   }
private:
  Fred* p;
};
```

```
int main()
{
    try
    {
      Foo* lptr = new Foo;
    }
    catch (...)
    {}
}
```

The consturctor for class foo would throw an exception for some random reason. I understand that the desturctor of foo will never be called but in this case will the destructor for p get called?

Question: What is Catching Class Types Discus with Example?

Catching Class Types:

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following
example demonstrates this.

```
// Catching class type exceptions.
#include <iostream>
#include <cstring>
using namespace std;
class MyException {
public:
char str_what[80];
int what;
MyException() { *str_what = 0; what = 0; }
MyException(char *s, int e) {
strcpy(str_what, s);
what = e;
}
};
int main()
{
int i;
try {
cout << "Enter a positive number: ";
cin >> i;
if(i<0)
throw MyException("Not Positive", i);
}
catch (MyException e) { // catch an error
cout << e.str_what << ": ";
cout << e.what << "\n";
}
return 0;
}
Here is a sample run:
Enter a positive number: -4
Not Positive: -4
```

The program prompts the user for a positive number. If a negative number is entered, an object of the class MyException is created that describes the error. Thus, MyException encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

Question: how you can Handling Derived –Class Exception Explain with example?

Handling Derived-Class Exceptions:

You need to be careful how you order your **catch** statements when trying to catch exception types that involve base and derived classes because a **catch** clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. If you don't do this, the base class **catch** will also catch all
derived classes.
For example, consider the following program.

```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main()
{
D derived;
try {
throw derived;
}
catch(B b) {
cout << "Caught a base class.\n";
}
catch(D d) {
cout << "This won't execute.\n";
}
return 0;
}
```

Here, because derived is an object that has B as a base class, it will be caught by the first catch clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the catch clauses.

Question: Describe the Exception Handling Option with Example?

Exception Handling Options:

There are several additional features and nuances to C++ exception handling that make it easier and more convenient to use. These attributes are discussed here.
Catching All Exceptions
In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch(...) {
// process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch(...)**.

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void Xhandler(int test)
{
try{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
catch(...) { // catch all exceptions
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
500 C + + : T h e C o m p l e t e R e f e r e n c e
return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```

As you can see, all three **throw**s were caught using the one **catch** statement. One very good use for **catch(...)** is as the last **catch** of a cluster of catches. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the preceding program explicity catches integer exceptions but relies upon **catch(...)** to catch all others.
// This example uses catch(...) as a default.

```
#include <iostream>
using namespace std;
void Xhandler(int test)
{
```

```
try{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
catch(int i) { // catch an int exception
cout << "Caught an integer\n";
}
catch(...) { // catch all other exceptions
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
C h a p t e r 1 9 : E x c e p t i o n H a n d l i n g 501
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
```

The output produced by this program is shown here.

```
Start
Caught an integer
Caught One!
Caught One!
End
```

As this example suggests, using **catch(...)** as a default is a good way to catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

Question: What are the Restriction In Exception ? Expalain with Example.

<u>Restricting Exceptions:</u>

You can restrict the type of exceptions that a function can throw outside of itself. In
fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a **throw** clause to a function definition. The general form of this is shown here:

```
ret-type func-name(arg-list) throw(type-list)
{
// ...
}
```

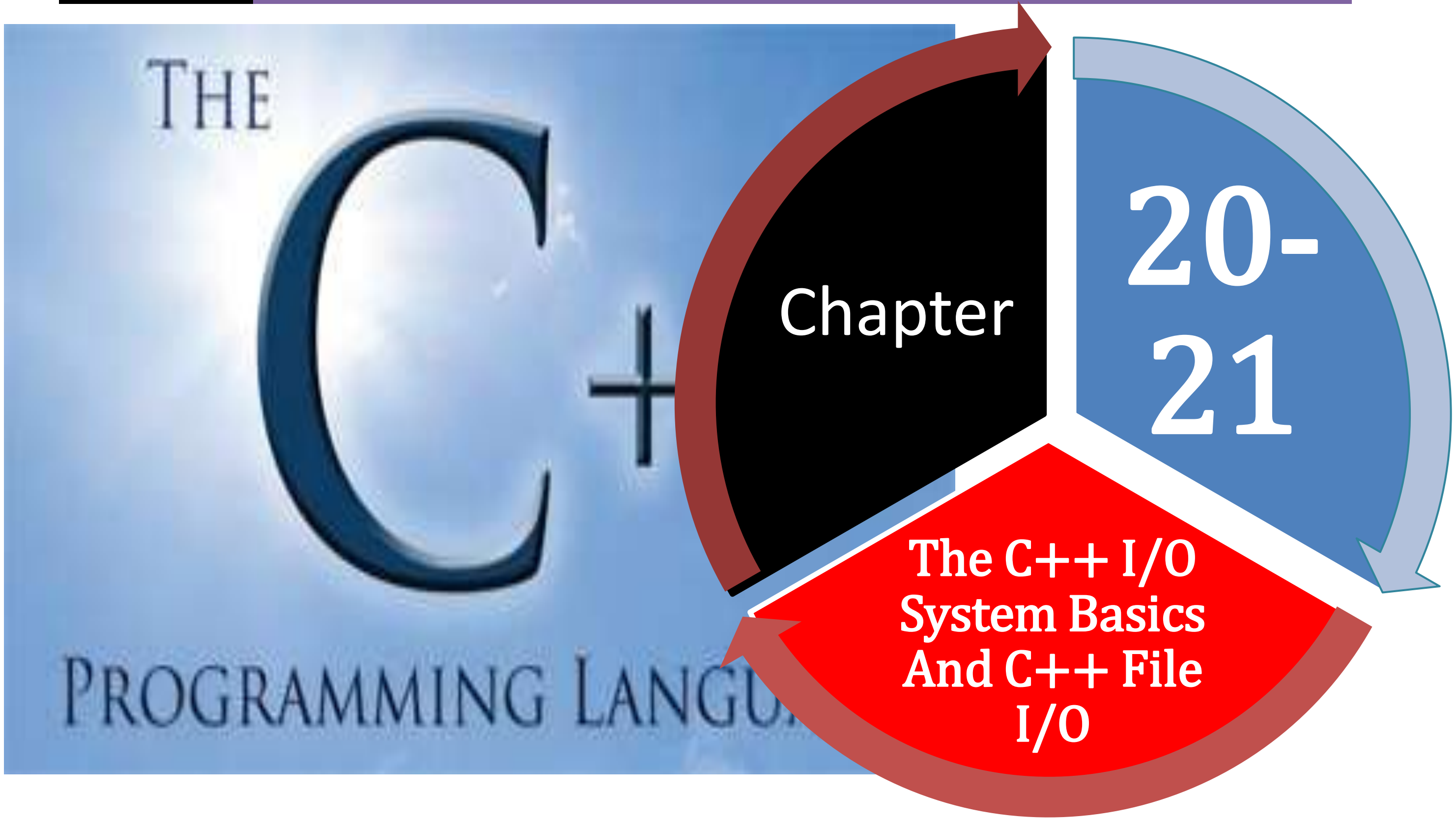
Here, only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, then use an empty list. Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected()** to be called. By default, this causes **abort()** to be called, which

causes abnormal program termination. However, you can specify your own unexpected handler if you like, as described later in this chapter. The following program shows how to restrict the types of exceptions that can be thrown from a function.
// Restricting function throw types.

```
#include <iostream>
using namespace std;
This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
int main()
{
cout << "start\n";
try{
Xhandler(0); // also, try passing 1 and 2 to Xhandler()
}
catch(int i) {
cout << "Caught an integer\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double d) {
cout << "Caught double\n";
}
cout << "end";
return 0;
}
```

In this program, the function **Xhandler()** may only throw integer, character, and **double** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, **unexpected()** will be called.) To see an example of this, remove **int** from the list and retry the program. It is important to understand that a function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block within a function may throw any type of exception so long as it is caught within that function. The restriction applies only when throwing an exception outside of the function. The following change to **Xhandler()** prevents it from throwing any exceptions.
// This function can throw NO exceptions!

```
void Xhandler(int test) throw()
{
/* The following statements no longer work. Instead,
they will cause an abnormal program termination. */
if(test==0) throw test;
if(test==1) throw 'a';
if(test==2) throw 123.23;
}
```



Important Questions

1. What is stream? Name the streams generally used file I/O. **Ex-2013**
2. Explain various file modes used in C++. **Ex-2013**
3. Explain the various file stream classes needed for file manipulations in C++. **Ex-2013**
4. Explain the functions seekg,seekp, tellg,tellp used for setting pointers during file operation. **Ex-2013**

Question:What is stream? Name the streams generally used file I/O. Ex-2013

Stream:

A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, you can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

Name The streams Generally used file I/O:

There are three types of streams:

- **Input,**
- **Output, and**
- **Input/output.**

To create an **input stream**, you must declare the stream to be of class **ifstream**.
To create an **output stream**, you must declare it as class **ofstream**.
Streams that will be performing **both input and output** operations must be declared as class **fstream.**

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes. Generally used I/O file are:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios.**

Question:What do you mean by Opening and Closing File Discuss whith Example?

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream or fstream** object may be used to open a file for writing and i**fstream** object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
ifstream in;   // input
ofstream out;  // output
fstream io;   // input and output
```

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.
You can combine two or more of these values by ORing them together.

Different Mode:

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File:

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.
Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void close();

Question:Explain various file modes used in C++. Ex-2013

File Mode:

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

Question:Explain the various file stream classes needed for file manipulations in C++. Ex-2013

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. This classes are called String Classes. These classes are declared in the header file iostream.This file should be included in all the programs that communicate with the console unit.

Stream classes for console I/O operations

ios is the base class for istream(input stream)and ostream(output stream) which are ,in turn, base classes for iostream(input/output stream).The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream provides the facilities for formatted and unformatted output. The class iostream provides the facility for handling both input and output streams.
Three classes,namely,istream_withassign,ostream_withassign and iostream_withassign add assignment operators these classes.

ios (General input/output stream class):

1. Contains basic facilities that are used by all other input and output classes.
2. Also contains a pointer to a buffer object(streambuf object).
3. Declares constants and functions that are necessary for handling formatted input and output operation.

istream(input stream):

1. Inherits the properties of ios
2. Declares input functions such as get(),getline() and read().
3. Contains overloaded extraction operator.

ostream(output stream):

1. Inherits the properties of ios
2. Declares output functions such as put() and write().
3. Contains overloaded initiation operator.

iostream(input/output stream):

- Inherits the properties of ios istream and ostream through multiple inheritance and thus contains all the input and output functions.

Streambuf:

- Provides an interface to physical devices through buffers.

Acts as a base for filebuf class used ios files

Question: What do you mean by Writing and Reading a file? Explain with give a Appropriate example?

Writing to a File:

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

For example, this program creates a short inventory file that contains each item's name and its cost:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream out("INVNTRY"); // output, normal file
if(!out) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
out << "Radios " << 39.95 << endl;
out << "Toasters " << 19.95 << endl;
out << "Mixers " << 24.80 << endl;
out.close();
return 0;
}
```

Reading from a File:

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("INVNTRY"); // input
if(!in) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
char item[20];
float cost;
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in.close();
return 0;
}
```

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
#include <iostream>
using namespace std;

int main () {

  char data[100];

  // open a file in write mode.
  ofstream outfile;
  outfile.open("afile.dat");

  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);

  // write inputted data into the file.
  outfile << data << endl;

  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();

  // again write inputted data into the file.
  outfile << data << endl;

  // close the opened file.
  outfile.close();

  // open a file in read mode.
  ifstream infile;
  infile.open("afile.dat");

  cout << "Reading from the file" << endl;
  infile >> data;

  // write the data at the screen.
  cout << data << endl;

  // again read the data from the file and display it.
  infile >> data;
  cout << data << endl;

  // close the opened file.
  infile.close();

  return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

Question: What are the task of put and get function discuss with example?

put( ) and get( ):

One way that you may read and write unformatted data is by using the member functions **get()** and **put()** . These functions operate on characters. That is, **get()** will read a character and **put()** will write a character. Of course, if you have opened the file for binary operations and are operating on a **char** (rather than a **wchar_t** stream), then these functions read and write bytes of data. The **get()** function has many forms, but the most commonly used version is shown here along with **put()** :

istream &get(char &ch); ostream &put(char ch);
The **get()** function reads a single character from the invoking stream and puts that value in ch. It returns a reference to the stream. The **put()** function writes ch to the stream and returns a reference to the stream.
The following program displays the contents of any file, wh ether it contains text or binary data, on the screen. It uses the **get()** function.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
char ch;
if(argc!=2) {
cout << "Usage: PR <filename>\n";
return 1;
}
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
cout << "Cannot open file.";
return 1;
}
while(in) { // in will be false when eof is reached
in.get(ch);
if(in) cout << ch;
}
return 0;
}
```

As stated in the preceding section, when the end-of-file is reached, the stream associated with the file becomes false. Therefore, when **in** reaches the end of the file, it will be false, causing the **while** loop to stop.
There is actually a more compact way to code the loop that reads and displays afile, as shown here:

```
while(in.get(ch))
cout << ch;
```

This works because **get()** returns a reference to the stream **in**, and **in** will be false when the end of the file is encountered.

Question: What do you mean by read and write function discuss with appropriate example?
<u>read( ) and write( ):</u>
Another way to read and write blocks of binary data is to use C++'s **read()** and **write()** functions. Their prototypes are

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

The **read()** function reads num characters from the invoking stream and puts them in the buffer pointed to by buf. The **write()** function writes num characters to the invoking stream from the buffer pointed to by buf. **streamsize** is a type defined by the C++ library as some form of integer. It is capable of holding the largest number of characters that can be transferred in any one I/O operation.
The program writes a structure to disk and then reads it back in:

```
}#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct status {
char name[80];
double balance;
unsigned long account_num;
};
int main()
{
struct status acc;
strcpy(acc.name, "Ralph Trantor");
acc.balance = 1123.23;
acc.account_num = 34235678;
// write data
ofstream outbal("balance", ios::out | ios::binary);
if(!outbal) {
cout << "Cannot open file.\n";
return 1;
}
outbal.write((char *) &acc, sizeof(struct status));
outbal.close();
// now, read back;
ifstream inbal("balance", ios::in | ios::binary);
if(!inbal) {
cout << "Cannot open file.\n";
return 1;

inbal.read((char *) &acc, sizeof(struct status));
cout << acc.name << endl;
cout << "Account # " << acc.account_num;
cout.precision(2);
cout.setf(ios::fixed);
cout << endl << "Balance: $" << acc.balance;
inbal.close();
return 0;
}
```

As you can see, only a single call to **read()** or **write()** is necessary to read or write the entire structure. Each individual field need not be read or written separately. As this example illustrates, the buffer can be any type of object. The type casts inside the calls to **read( )** and

write( ) are necessary when operating on a buffer that is not defined as a character array. Because of C++'s strong type checking, a pointer of one type will not automatically be converted into a pointer of another type.

Question: Is it possible Detecting EOF ? if possible how ? discuss with example?

Detecting EOF:

You can detect when the end of the file is reached by using the member function **eof(),** which has this prototype:

bool eof( );

It returns true when the end of the file has been reached; otherwise it returns false. The following program uses **eof()** to display the contents of a file in both hexadecimal and ASCII.

Display contents of specified file in both ASCII and in hex.

```
#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: Display <filename>\n";
return 1;
}
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
cout << "Cannot open input file.\n";
return 1;
}
register int i, j;
int count = 0;
char c[16];
cout.setf(ios::uppercase);
while(!in.eof()) {
for(i=0; i<16 && !in.eof(); i++) {
in.get(c[i]);
}
if(i<16) i--; // get rid of eof
for(j=0; j<i; j++)
cout << setw(3) << hex << (int) c[j];
for(; j<16; j++) cout << " ";
cout << "\t";
for(j=0; j<i; j++)
if(isprint(c[j])) cout << c[j];
else cout << ".";
cout << endl;
count++;
if(count==16) {
```

```
count = 0;
cout << "Press ENTER to continue: ";
cin.get();
cout << endl;
}
}
in.close();
return 0;
}
```

Question:Explain the functions seekg,seekp, tellg,tellp used for setting pointers during file operation. Ex-2013

Answer:

Random Access:

In C++'s I/O system, you perform random access by using the seekg() and seekp() functions. Their most common forms are

istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);

Here, off_type is an integer type defined by ios that is capable of containing the largest valid value that offset can have. seekdir is an enumeration defined by ios that determines how the seek will take place.

using the seekg() and seekp() functions allows you to access the file in a nonsequential fashion.

Seekg() function:

The seekg() function moves the associated file's current get pointer offset number of characters from the specified origin, which must be one of these three values:

ios::beg	Beginning-of-file
ios::cur	Current location
ios::end	End-of-file

seekp() function:

The seekp() function moves the associated file's current put pointer offset number of characters from the specified origin, which must be one of the values just shown. Generally, random-access I/O should be performed only on those files opened for binary operations. The character translations that may occur on text files could cause a position request to be out of sync with the actual contents of the file.

Obtaining the Current File Position:

Tellg() and tellp():

You can determine the current position of each file pointer by using these functions:

pos_type tellg( );
pos_type tellp( );

Here, pos_type is a type defined by ios that is capable of holding the largest value that either function can return. **You can use the values returned by tellg() and tellp() as arguments to the following forms of seekg() and seekp() , respectively.**

```
istream &seekg(pos_type pos);
ostream &seekp(pos_type pos);
```

These functions allow you to save the current file location, perform other file operations, and then reset the file location to its previously saved location.

Reference:

[1].

About the Authors

Abu Saleh Musa Miah(abid) was born in Rangpur,Bangladesh in 1992. He received the B.Sc. Engineering degree with Honours in Computer science and Engineering in 2014 with FIRST merit Position Engineering first batch from University of Rajshahi ,Bangladesh. He also completed research on the field **COMPUTER VISION** specifically Moving Object Detection with BoofCV tools(java). Currently he is working towards the M.Sc.Engineering degree in the Department of Computer Science And Engineering University of Rajshahi, Bangladesh.

His research interests include **Brain Computer Interfacing**. Sparse signal recovery/compressed sensing, blind source separation, neuroimaging and computational and cognitive neuroscience.

Email: abusalehcse.ru@gmail.com;

https://www.facebook.com/abusalehmusa.miah