**BCS-031 C++ JULY 2014.docx**

1. **(a) What is Object Oriented Programming? Explain its features with example. (5 Marks)**

**Object-oriented programming** (**OOP**) is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) that represents the concept of "[objects](http://en.wikipedia.org/wiki/Object_(computer_science))" that have [data fields](http://en.wikipedia.org/wiki/Field_(computer_science))(attributes that describe the object) and associated procedures known as [methods](http://en.wikipedia.org/wiki/Method_(computer_science)). Objects, which are usually [instances](http://en.wikipedia.org/wiki/Instance_(computer_science)) of [classes](http://en.wikipedia.org/wiki/Class_(computer_science)), are used to interact with one another to design applications and computer programs [C++](http://en.wikipedia.org/wiki/C%2B%2B), [Objective-C](http://en.wikipedia.org/wiki/Objective-C), [Smalltalk](http://en.wikipedia.org/wiki/Smalltalk), [Delphi](http://en.wikipedia.org/wiki/Delphi_(programming_language)), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)),[Javascript](http://en.wikipedia.org/wiki/Javascript), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [Perl](http://en.wikipedia.org/wiki/Perl), [Python](http://en.wikipedia.org/wiki/Python_(programming_language)), [Ruby](http://en.wikipedia.org/wiki/Ruby_(programming_language)) and [PHP](http://en.wikipedia.org/wiki/PHP) are examples of object-oriented programming languages.

Object-oriented programming attempts to provide a model for programming based on objects.[[3]](http://en.wikipedia.org/wiki/Object-oriented_programming) Object-oriented programming integrates code and data using the concept of an "object". An object is an [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) with the addition of [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_(computer_science)) and [inheritance](http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)). An object has both state (data) and behavior (code).

Characteristics of the Object Oriented programming are:

* Emphasis on data rather than procedure
* Programs are divided into entities known as objects
* Data Structures are designed such that they characterize objects
* Functions that operate on data of an object are tied together in data structures
* Data is hidden and cannot be accessed by external functions
* Objects communicate with each other through functions
* New data and functions can be easily added whenever necessary
* Follows bottom up design in program design

**Features of OOP**

OOP stands for Object Oriented Programming and the language that support this Object Oriented programming features is called Object oriented Programming Language. An example of a language that support this Object oriented features is C++.

**Features of Object oriented Programming**

The Objects Oriented programming language supports all the features of normal programming languages. In addition it supports some important concepts and terminology which has made it popular among programming methodology.   
  
The important features of Object Oriented programming are:

* Inheritance
* Polymorphism
* Data Hiding
* Encapsulation
* Overloading
* Reusability

**(b) Write a C++ program to create Matrix class. This class should have functions to find the sum and difference of two matrices. (9 Marks)**

#include<iostream.h>

#include<conio.h>

Class Matrix

{

int a[3][3],b[3][3],i,j;

Public:void Input()

{

//Input the numbers of first matrix

cout<<"First Matrix"<<endl;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

cout<<"Enter number :";

cin>>a[i][j];

}

}

//Input the numbers of second matrix

cout<<"Second Matrix"<<endl;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

cout<<"Enter number :";

cin>>b[i][j];

}

}

}

Void sum()

{

//display the sum of matrices

cout<<"Sum is"<<endl;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

cout<<a[i][j]+b[i][j]<<"t";

}

cout<<endl;

}

}

Void Difference()

{

//display the Difference of matrices

cout<<"Difference is"<<endl;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

cout<<a[i][j]-b[i][j]<<"t";

}

cout<<endl;

}

}

}

void main()

{

clrscr();

Matrix M1;

M1.input();

M1.sum();

M1.Difference();

getch();

}

**(c) Explain the usage of the following C++ operators with the help of an example program. (6 Marks)**

(a) Relational Operator

(b) Logical Operators

(c) Scope resolution operator

*Ans:-*

**(a) Relational Operators:**

There are following relational operators supported by C++ language

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

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

Example:-

#include <iostream>

using namespace std;

main()

{

int a = 21;

int b = 10;

int c ;

if( a == b )

{

cout << "Line 1 - a is equal to b" << endl ;

}

else

{

cout << "Line 1 - a is not equal to b" << endl ;

}

if ( a < b )

{

cout << "Line 2 - a is less than b" << endl ;

}

else

{

cout << "Line 2 - a is not less than b" << endl ;

}

if ( a > b )

{

cout << "Line 3 - a is greater than b" << endl ;

}

else

{

cout << "Line 3 - a is not greater than b" << endl ;

}

/\* Let's change the values of a and b \*/

a = 5;

b = 20;

if ( a <= b )

{

cout << "Line 4 - a is either less than \

or euqal to b" << endl ;

}

if ( b >= a )

{

cout << "Line 5 - b is either greater than \

or equal to b" << endl ;

}

return 0;

}

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

Line 1 - a is not equal to b

Line 2 - a is not less than b

Line 3 - a is greater than b

Line 4 - a is either less than or euqal to b

Line 5 - b is either greater than or equal to b

**(b) Logical Operators:**

There are following logical operators supported by C++ language

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

[Show Examples](http://www.tutorialspoint.com/cplusplus/cpp_logical_operators.htm)

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

Example:-

#include <iostream>

using namespace std;

main()

{

int a = 5;

int b = 20;

int c ;

if ( a && b )

{

cout << "Line 1 - Condition is true"<< endl ;

}

if ( a || b )

{

cout << "Line 2 - Condition is true"<< endl ;

}

/\* Let's change the values of a and b \*/

a = 0;

b = 10;

if ( a && b )

{

cout << "Line 3 - Condition is true"<< endl ;

}

else

{

cout << "Line 4 - Condition is not true"<< endl ;

}

if ( !(a && b) )

{

cout << "Line 5 - Condition is true"<< endl ;

}

return 0;

}

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

Line 1 - Condition is true

Line 2 - Condition is true

Line 4 - Condition is not true

Line 5 - Condition is true

**(c) Scope resolution operator**

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

C++ programming code

#include <iostream>

using namespace std;

char c = 'a'; // global variable

int main() {

char c = 'b'; //local variable

cout << "Local c: " << c << "\n";

cout << "Global c: " << ::c << "\n"; //using scope resolution operator

return 0;

}

**Scope resolution operator in class**

#include <iostream>

using namespace std;

class programming {

public:

void output(); //function declaration

};

// function definition outside the class

void programming::output() {

cout << "Function defined outside the class.\n";

}

int main() {

programming x;

x.output();

return 0;

}

**Question 2:**

**(a) Define the class Teacher with all the basic attributes such as Name, Department,Subjects,date\_of\_ joining, years\_of\_experience etc. Define constructor(s), member functions display\_detail() for displaying the Teacher details. Use appropriate access control specifiers in this program. Also inherit Post\_Graduate\_Teacher from Teacher class.(9 Marks)**

*Ans:-*

Class Teacher

{

Int TeacherID;

Char Tname[20];

Char Fathernm[20];

Char Dept[20];

Char JoiningDate[10];

Int Experience;

Public: Teacher(int TID,Char tnm[ ], Char Fnm[ ], Char D[ ], Char JD[ ], int e)

{

TeacherID=TID;

Strcpy(Tname,tnm);

Strcpy(Fathernm,Fnm);

Strcpy(Dept,D);

Strcpy(JoiningDate,JD);

Experience=e;

}

Public: void display\_details()

{

Cout<<”Teacher ID=”<<TeacherID<<endl;

Cout<<”Teacher Name=”<<Tname<<endl;

Cout<<”Father Name=”<<Fathernm<<endl;

Cout<<”Department=”<<Dept<<endl;

Cout<<”Joining Date=”<<JoiningDate<<endl;

Cout<<”Years Of Experience=”<<Experience<<endl;

}

}

Class Post\_Graduate\_Teacher : public Teacher

{

Char Qual[10];

Public: Post\_Graduate\_Teacher(int TID,Char tnm[ ], Char Fnm[ ], Char D[ ], Char JD[ ], int e,Char Q[ ])

{

Teacher(TID,tnm,Fnm,D,JD,e);

Strcpy(Qual,Q);

}

Void ShowDetails()

{

Display\_details();

Cout<<”Qualification=”<<Qual<<endl;

}

}

Void main()

{

Post\_Graduate\_Teacher PGT(101,”Amit”,”Mr.Kumar”,”Maths”,”2/2/02”,6,“PhD”);

PGT.showdetails();

}

**(b) Explain the following terms in the context of object oriented programming. Also explain how these concepts are implemented in C++ by giving an example program for each. (6 Marks)**

**(a) Virtual Function**

**(b) Operator Overloading**

*Ans:-*

**(a)Virtual Function**

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function’s declaration in the base class with the keyword virtual. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

Base class pointer can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked. But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.

Consider following program code:

Class A   
{   
        int a;   
        public:   
        A()   
        {  
                 a = 1;   
        }   
        virtual void show()   
        {   
                    cout <<a;  
        }   
};

Class B: public A   
{   
         int b;   
         public:   
         B()   
         {   
                 b = 2;   
         }  
         virtual void show()   
         {    
                  cout <<b;  
         }  
};

int main()   
{   
           A \*pA;   
           B oB;   
           pA = &oB;   
           pA->show();   
           return 0;   
}

Output is 2 since pA points to object of B and show() is virtual in base class A.

**(b) Operator overloading**

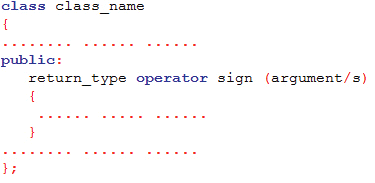
The meaning of operators are already defined and fixed for basic types like: int, float, double etc in C++ language. For example: If you want to add two integers then, + operator is used. But, for user-defined types(like: objects), you can define the meaning of operator, i.e, you can redefine the way that operator works. For example: If there are two objects of a class that contain string as its data member, you can use + operator to concatenate two strings. Suppose, instead of strings if that class contains integer data member, then you can use + operator to add integers. This feature in C++ programming that allows programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading.

**Why Operator overloading is used in C++ programming?**

You can write any C++ program without the knowledge of operator overloading. But, operator operating are profoundly used by programmer to make a program clearer. For example: you can replace the code like: **calculation = add(mult(a,b),div(a,b));** with **calculation = a\*b+a/b;** which is more readable and easy to understand.

**How to overload operators in C++ programming?**

To overload a operator, a operator function is defined inside a class as:



The return type comes first which is followed by keyword **operator**, followed by operator sign,i.e., the operator you want to overload like: +, <, ++ etc. and finally the arguments is passed. Then, inside the body of you want perform the task you want when this operator function is called.

This operator function is called when, the operator(sign) operates on the object of that class class\_name.

**Example of operator overloading in C++ Programming**

/\* Simple example to demonstrate the working of operator overloading\*/

#include <iostream>

using namespace std;

class temp

{

private:

int count;

public:

temp():count(5){ }

void operator ++() {

count=count+1;

}

void Display() { cout<<"Count: "<<count; }

};

int main()

{

temp t;

++t; /\* operator function void operator ++() is called \*/

t.Display();

return 0;

}

**Output**

**Count: 6**

**Explanation**

In this program, a operator function **void operator ++ ()** is defined(inside class *temp*), which is invoked when ++ operator operates on the object of type *temp*. This function will increase the value of count by 1.

**Question 3:**

**(a) What is polymorphism? What are different forms of polymorphism? Explain implementation of polymorphism with the help of a C++ program. (8 Marks)**

In [programming languages](http://en.wikipedia.org/wiki/Programming_language) and [type theory](http://en.wikipedia.org/wiki/Type_theory), **polymorphism** is the provision of a single [interface](http://en.wikipedia.org/wiki/Interface_(computing)) to entities of different [types](http://en.wikipedia.org/wiki/Data_type). A **polymorphic type** is a type whose operations can also be applied to values of some other type, or types.

There are several fundamentally different kinds of polymorphism:

* If a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations, it is called *[adhoc](http://en.wikipedia.org/wiki/Ad_hoc_polymorphism)*[polymorphism](http://en.wikipedia.org/wiki/Ad_hoc_polymorphism). *Ad hoc* polymorphism is supported in many languages using [function overloading](http://en.wikipedia.org/wiki/Function_overloading).
* If the code is written without mention of any specific type and thus can be used transparently with any number of new types, it is called [parametric polymorphism](http://en.wikipedia.org/wiki/Parametric_polymorphism). In the object-oriented programming community, this is often known as *generics* or [*generic programming*](http://en.wikipedia.org/wiki/Generic_programming). In the functional programming community, this is often simply called *polymorphism*.
* [Subtyping](http://en.wikipedia.org/wiki/Subtyping) (or *inclusion polymorphism*) is a concept wherein a name may denote instances of many different classes as long as they are related by some common superclass.[[3]](http://en.wikipedia.org/wiki/Polymorphism_(computer_science)) In object-oriented programming, this is often referred to simply as *polymorphism*.

#include<iostream.h>

#include<conio.h>

Class base

{

public:

void show() { cout<<"base class"<<endl; }

};

Class derv\_1:public base

{

public:

void show() { cout<<"derv\_1 class"<<endl; }

};

class derv\_2:publicbase

{

public:

void show() { cout<<"derv\_2 class"<<endl; }

};

main()

{

clrscr();

base b\_obj;

b\_obj.show();

derv\_1 d\_1\_obj;

derv\_2 d\_2\_obj;

d\_1\_obj.show();

d\_2\_obj.show();

getch();

return 0;

}

**(b) What is access control specifier ? Explain the need of different access control specifiers withexample. (7 Marks)**

*Ans:-*

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

class Base {

public:

// public members go here

protected:

// protected members go here

private:

// private members go here

};

**The public members:**

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 as shown in the following example:

#include <iostream>

using namespace std;

class Line

{

public:

double length;

void setLength( double len );

double getLength( void );

};

// Member functions definitions

double Line::getLength(void)

{

return length ;

}

void Line::setLength( double len )

{

length = len;

}

// Main function for the program

int main( )

{

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

// set line length without member function

line.length = 10.0; // OK: because length is public

cout << "Length of line : " << line.length <<endl;

return 0;

}

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

Length of line : 6

Length of line : 10

**The private members:**

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 );

};

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

#include <iostream>

using namespace std;

class Box

{

public:

double length;

void setWidth( double wid );

double getWidth( void );

private:

double width;

};

// Member functions definitions

double Box::getWidth(void)

{

return width ;

}

void Box::setWidth( double wid )

{

width = wid;

}

// Main function for the program

int main( )

{

Box box;

// set box length without member function

box.length = 10.0; // OK: because length is public

cout << "Length of box : " << box.length <<endl;

// set box width without member function

// box.width = 10.0; // Error: because width is private

box.setWidth(10.0); // Use member function to set it.

cout << "Width of box : " << box.getWidth() <<endl;

return 0;

}

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

Length of box : 10

Width of box : 10

**The protected members:**

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.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox** from a parent class **Box**.

Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox.

#include <iostream>

using namespace std;

class Box

{

protected:

double width;

};

class SmallBox:Box // SmallBox is the derived class.

{

public:

void setSmallWidth( double wid );

double getSmallWidth( void );

};

// Member functions of child class

double SmallBox::getSmallWidth(void)

{

return width ;

}

void SmallBox::setSmallWidth( double wid )

{

width = wid;

}

// Main function for the program

int main( )

{

SmallBox box;

// set box width using member function

box.setSmallWidth(5.0);

cout << "Width of box : "<< box.getSmallWidth() << endl;

return 0;

}

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

**Question 4 :**

**(a) Explain the concept of copy constructor with the help of an example program. (4 Marks)**

*Ans:-*

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old\_obj);

Following is a simple example of copy constructor.

|  |
| --- |
| #include<iostream>  using namespace std;    class Point  {  private:      int x, y;  public:      Point(int x1, int y1) { x = x1; y = y1; }        // Copy constructor      Point(const Point &p2) {x = p2.x; y = p2.y; }        int getX()            {  return x; }      int getY()            {  return y; }  };    int main()  {      Point p1(10, 15); // Normal constructor is called here      Point p2 = p1; // Copy constructor is called here        // Let us access values assigned by constructors      cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();      cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();        return 0;  } |

Output:

p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15

**When is copy constructor called?**  
In C++, a Copy Constructor may be called in following cases:  
1. When an object of the class is returned by value.  
2. When an object of the class is passed (to a function) by value as an argument.  
3. When an object is constructed based on another object of the same class.  
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the [return value optimization (sometimes referred to as RVO)](http://en.wikipedia.org/wiki/Return_value_optimization).

**(b) What is an exception? How an exception is different from an error? Explain advantage of exceptions handling in C++, with the help of an example program. (6 Marks)**

*Ans:-*

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

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

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

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

**Throwing Exceptions:**

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

double division(int a, int b)

{

if( b == 0 )

{

throw "Division by zero condition!";

}

return (a/b);

}

**Catching Exceptions:**

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

try

{

// protected code

}catch( ExceptionName e )

{

// code to handle ExceptionName exception

}

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

try

{

// protected code

}catch(...)

{

// code to handle any exception

}

The following is an example, which throws a division by zero exception and we catch it in catch block.

#include <iostream>

using namespace std;

double division(int a, int b)

{

if( b == 0 )

{

throw "Division by zero condition!";

}

return (a/b);

}

int main ()

{

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

}catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

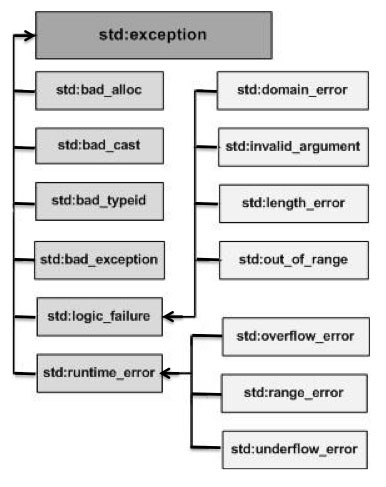
}

Because we are raising an exception of type **const char\***, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

**C++ Standard Exceptions:**

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

|  |  |
| --- | --- |
| **Exception** | **Description** |
| **std::exception** | An exception and parent class of all the standard C++ exceptions. |
| std::bad\_alloc | This can be thrown by **new**. |
| std::bad\_cast | This can be thrown by **dynamic\_cast**. |
| std::bad\_exception | This is useful device to handle unexpected exceptions in a C++ program |
| std::bad\_typeid | This can be thrown by **typeid**. |
| **std::logic\_error** | An exception that theoretically can be detected by reading the code. |
| std::domain\_error | This is an exception thrown when a mathematically invalid domain is used |
| std::invalid\_argument | This is thrown due to invalid arguments. |
| std::length\_error | This is thrown when a too big std::string is created |
| std::out\_of\_range | This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[](). |
| **std::runtime\_error** | An exception that theoretically can not be detected by reading the code. |
| std::overflow\_error | This is thrown if a mathematical overflow occurs. |
| std::range\_error | This is occured when you try to store a value which is out of range. |
| std::underflow\_error | This is thrown if a mathematical underflow occurs. |

**Define New Exceptions:**

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception

{

const char \* what () const throw ()

{

return "C++ Exception";

}

};

int main()

{

try

{

throw MyException();

}

catch(MyException& e)

{

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

}

catch(std::exception& e)

{

//Other errors

}

}

This would produce the following result:

MyException caught

C++ Exception

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

**(c) What is data stream? Explain stream hierarchy in C++. (5 Marks)**

*Ans:-*

One of the great strengths of C++ is its I/O system, IO Streams. As Bjarne Stroustrup says in his book "The C++ Programming Language", "Designing and implementing a general input/output facility for a programming language is notoriously difficult". He did an excellent job, and the C++ IOstreams library is part of the reason for C++'s success. IO streams provide an incredibly flexible yet simple way to design the input/output routines of any application.

Streams are serial interfaces to storage, buffers files, or any other storage medium. The difference between storage media is intentionally hidden by the interface; you may not even know what kind of storage you're working with but the interface is exactly the same.

The "serial" nature of streams is a very important element of their interface. You cannot directly make random access random reads or writes in a stream (unlike, say, using an array index to access any value you want) although you can seek to a position in a stream and perform a read at that point.

IOstreams can be used for a wide variety of data manipulations thanks to the following features:

* A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar\_t). Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation.
* Streams work with built-in data types, and you can make user-defined types work with streams by [overloading](http://www.cprogramming.com/tutorial/operator_overloading.html) the insertion operator (<<) to put objects into streams, and the extraction operator (>>) to read objects from streams.
* The stream library's unified approach makes it very friendly to use. Using a consistent interface for outputting to the screen and sending files over a network makes life easier. The programs below will show you what is possible.

HIERARCHY OF STREAMS

C++ provides the following classes to perform output and input of characters to/from files:

* [**ofstream**](http://www.cplusplus.com/ofstream)**:** Stream class to write on files
* [**ifstream**](http://www.cplusplus.com/ifstream)**:** Stream class to read from files
* [**fstream**](http://www.cplusplus.com/fstream)**:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | // basic file operations  #include <iostream>  #include <fstream>  using namespace std;  int main () {  ofstream myfile;  myfile.open ("example.txt");  myfile << "Writing this to a file.\n";  myfile.close();  return 0;  } | [file example.txt]  Writing this to a file. | [Edit & Run](http://www.cplusplus.com/doc/tutorial/files/) |

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.  
  
But let's go step by step:

**Open a file**

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a *stream* (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.  
  
In order to open a file with a stream object we use its member function open:  
  
open (filename, mode);  
  
Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

|  |  |
| --- | --- |
| ios::in | Open for input operations. |
| ios::out | Open for output operations. |
| ios::binary | Open in binary mode. |
| ios::ate | Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file. |
| ios::app | All output operations are performed at the end of the file, appending the content to the current content of the file. |
| ios::trunc | If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one. |

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.binin binary mode to add data we could do it by the following call to member function open:

|  |  |  |
| --- | --- | --- |
| 1 2 | ofstream myfile;  myfile.open ("example.bin", ios::out | ios::app | ios::binary); |  |

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

|  |  |
| --- | --- |
| **class** | **default mode parameter** |
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in | ios::out |

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).  
  
For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.  
  
File streams opened in *binary mode* perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).  
  
Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

|  |  |  |
| --- | --- | --- |
|  | ofstream myfile ("example.bin", ios::out | ios::app | ios::binary); |  |

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is\_open. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or falseotherwise:

|  |  |  |
| --- | --- | --- |
|  | if (myfile.is\_open()) { /\* ok, proceed with output \*/ } |  |

**Closing a file**

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file:

|  |  |  |
| --- | --- | --- |
|  | myfile.close(); |  |

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.  
  
In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

**Question 5:**

**(a ) What is template? Explain advantage of using template in C++? Write C++ program to explain function template and class template. (7 Marks)**

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 do 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.

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

**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

You can use templates to:

* Create a typesafe collection class (for example, a stack) that can operate on data of any type.
* Add extra type checking for functions that would otherwise take **void** pointers.
* Encapsulate groups of operator overrides to modify type behavior (such as smart pointers).

**Advantages:**

* Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
* Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
* Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

**(b) What is inheritance? Explain the different types of inheritance supported by C++? Explain whether constructors are inherited by derived class in C++ or not, write a program in support of your claim and show the output. (8 Marks)**

*Ans:-*

**Inheritance**

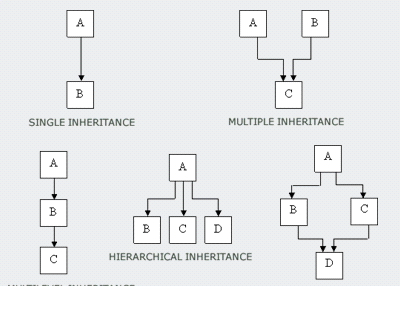
The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.

Inheritance lets you create new classes from existing class. Any new class that you create from an existing class is called **derived class**; existing class is called **base class**.

The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

**Forms of Inheritance**

**Single Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from one base class.  
**Multiple Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)  
**Hierarchical Inheritance:** It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.  
**Multilevel Inheritance:** It is the inheritance hierarchy wherein subclass acts as a base class for other classes.  
**Hybrid Inheritance:** The inheritance hierarchy that reflects any legal combination of other four types of inheritance.



In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format :

class derived\_class: memberAccessSpecifier base\_class

{

...

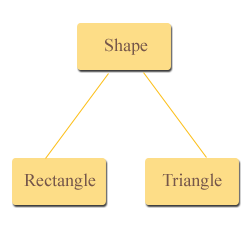
};

Where derived\_class is the name of the derived class and base\_class is the name of the class on which it is based. The member Access Specifier may be public, protected or private. This access specifier describes the access level for the members that are inherited from the base class.

|  |  |
| --- | --- |
| **Member Access Specifier** | **How Members of the Base Class Appear in the Derived Class** |
| Private | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become private members of the derived class. |
| Public members of the base class become private members of the derived class. |
| Protected | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become protected members of the derived class. |
| Public members of the base class become protected members of the derived class. |
| Public | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become protected members of the derived class. |
| Public members of the base class become public members of the derived class. |

In principle, a derived class inherits every member of a base class except constructor and destructor. It means private members are also become members of derived class. But they are inaccessible by the members of derived class.

Following example further explains concept of inheritance :



class Shape

{

protected:

float width, height;

public:

void set\_data (float a, float b)

{

width = a;

height = b;

}

};

class Rectangle: public Shape

{

public:

float area ()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public:

float area ()

{

return (width \* height / 2);

}

};

int main ()

{

Rectangle rect;

Triangle tri;

rect.set\_data (5,3);

tri.set\_data (2,5);

cout << rect.area() << endl;

cout << tri.area() << endl;

return 0;

}

|  |
| --- |
| output :  15 5 |

The object of the class Rectangle contains :  
width, height inherited from Shape becomes the protected member of Rectangle.  
set\_data() inherited from Shape becomes the public member of Rectangle  
area is Rectangle’s own public member  
  
The object of the class Triangle contains :  
width, height inherited from Shape becomes the protected member of Triangle.  
set\_data() inherited from Shape becomes the public member of Triangle  
area is Triangle’s own public member

set\_data () and area() are public members of derived class and can be accessed from outside class i.e. from main()