14BECS305 Object-Oriented Programming

L	T	P	C
3	0	0	3

COURSE OBJECTIVES:

- Understand the concepts of object-oriented, event driven, and concurrent Programming paradigms .
- Develop skills in using these paradigms using Java.
- Analyze and compare the efficiency of algorithms.
- Possess the ability to design efficient algorithms for solving computing problems.

LEARNING OUTCOMES:

- Able to use a simple Java programming environment, compile programs and interpret compiler errors.
- Able to understand and use the fundamental data types.
- Able to design classes and organise them into packages.
- Able to test programs to ensure that they perform as intended.

UNIT I Fundamentals of Object-Oriented Programming
(9)

Introduction to Object oriented programming – Benefits and Applications of OOP- structural programming versus object oriented programming - Simple Java Program - Data Types – Operators – Expressions - Decision Making and Loop control Statements - The? : Operator - Arrays-Strings – Getting input in java.

UNIT II Classes, Objects and Methods
(9)

Defining a Class-Creating Objects-Accessing Class Members-Constructors-Methods Overloading-Static Members-Nesting of Methods-Final Variables and Methods- Final Classes-Finalize Methods-Visibility Control

UNIT III Inheritance and Interfaces
(9)
Motivation - Inheritance: Extending a Class – Types of Inheritance - Overriding Methods - Interfaces in Java (Interface and Implement) - Multiple inheritance – Examples

UNIT IV Managing Errors and Exception Handling
(9)
Motivation – Exception handling – Exception hierarchy – Throwing and Catching exceptions - Syntax of Exception Handling Code - Types of Errors -Multiple Catch Statements - Using Finally Statement -User defined Exceptions - Using Exceptions for Debugging.

UNIT V Input /Output Streams
(9)

Motivation - I/O Streams - Concept of Streams- Stream Classes- Byte Stream Classes-Character Stream Classes-Using Streams-Other Useful I/O Classes- Using the File Class- Input /Output Exceptions-Creation of Files-Reading/Writing Characters-Reading/Writing Bytes - Handling Primitive Data Types - Concatenating and Buffering Files-Random Access Files-Interactive Input and Output-Other Stream classes.

Total Hours: 45

TEXT BOOKS:

1. Herbert Schildt "Java: The Complete Reference", 9th Edition, Mcgraw-Hill, 2014.
2. D.T. Editorial Services ,"Java 8 Programming: Black Book",Dreamtech Press, 2015.
3. Yashawant Kanetkar, "Let Us Java", 1st Edition, PBP Publications, 2012 .
4. C. Thomas Wu, "An Introduction to Object-Oriented programming with Java", 5th Edition Tata McGraw-Hill Publishing company Ltd 2010.

REFERENCES:

1. Cay S. Horstmann and Gary Cornel, "Core Java: Volume I – Fundamentals", 8th Edition, Sun
   Microsystems Press, 2011
2. Timothy Budd "Understanding Object-oriented programming with Java" Pearson Education,2nd edition, 2006
3. Herbert Schildt, "Java The Complete Reference", Oracle Press, 8th edition, 2011

WEBSITES:

1. http://java.sun.com.

Name of the Subject : OOP **SUBJECT CODE:14BECS305**

Class **: BE CSE**

S.No.	Duration	Topic Name	Teaching Aids	Page no. of Tex book
Unit - I - Fundamentals of Object-Oriented Programming				
1	1	Introduction to Object oriented programming	BB	T1->3-8
2	1	Benefits of OOP-Applications of OOP	VIDEO LEC & PPT	T2->7-8
3	1	Structural programming versus object oriented programming	PPT	W1
4	1	Tutorial : OOP Concepts	-	
5	1	Simple Java Program	BB	T2->24-26
6	1	Data Types – Operators	PPT	T1->33-39
7	1	Expressions - Decision Making and Loop control Statements	PPT	T1->77-103
8	1	*Tutorial - Datatypes and operators*	-	
9	1	The? : Operator - Arrays-Strings – Getting input in java	BB	T1->48-75
Unit - II - Classes, Objects and Methods				
10	1	Defining a Class-Creating Objects-Accessing Class Members	EXE	T1->105-109
11	1	Constructors	BB	T1->117-118
12	1	Methods Overloading	PPT	T1->125-128
13	1	*Tutorial - method overloading*	-	
14	1	Static Members-Nesting of Methods	PPT	T1->141-143
15	1	Final Variables and Methods-Final Classes	PPT	T1->182-183
16	1	Finalize Methods	EXE	T1->121-122
17	1	*Tutorial - Static Members*	-	
18	1	Visibility Control	PPT	T1->138
Unit - III - Inheritance and Interfaces				
19	1	Inheritance - Motivation	PPT	T1->159
20	1	Inheritance: Extending a Class	EXE	T2->176
21	1	Types of Inheritance	EXE	T1->177
22	1	*Tutorial - Inheritance*	-	
23	1	Overriding Methods	PPT	T1->173

24	1	Interfaces in Java (Interface and Implement)	PPT	T1->194-196
25	1	Multiple inheritance	EXE	T1->179-180
26	1	*Tutorial - Interfaces in Java*	-	
27	1	Inheritance Examples	BB	T1->178
Unit - IV -Managing Errors and Exception Handling				
28	1	Motivation - Exception handling	EXE	T1->205
29	1	Exception hierarchy	PPT	T1->206
30	1	Throwing and Catching exceptions	EXE	T1->208
31	1	*Tutorial - Throwing and Catching exceptions*	-	
32	1	Syntax of Exception Handling Code	EXE	T1->205-206
33	1	Types of Errors -Multiple Catch Statements	EXE	T1->221
34	1	Using Finally Statement - User defined Exceptions	BB	T1->216
35	1	*Tutorial -Using Finally Statement*	-	
36	1	Using Exceptions for Debugging	EXE	T1->-219
Unit - V - Input /Output Streams				
37	1	Motivation - I/O Streams - Concept of Streams	BB	T1->285
38	1	Stream Classes- Byte Stream Classes- Character Stream Classes	BB	T1->286
39	1	Using Streams-Other Useful I/O Classes	BB	T1->287-288
40	1	*Tutorial - I/O Streams*	-	
41	1	Using the File Class- Input /Output Exceptions-Creation of Files	BB	T1->290-292
42	1	Reading/Writing Characters-Reading/Writing Bytes	BB	T1->293
43	1	Concatenating and Buffering Files-Random Access Files	BB	T1->568-569
44	1	*Tutorial - Inheritance*	-	
45	1	Interactive Input and Output-Other Stream classes.	BB	T1->286-288
46	1	Previous year question papers discussion	-	

Total Hours allocated : 46

TEXT BOOKS:

1. Herbert Schildt "Java: The Complete Reference", 9th

Edition, Mcgraw-Hill, 2014.
2. D.T. Editorial Services ,"Java 8 Programming: Black Book",Dreamtech Press, 2015.
3. Yashawant Kanetkar, "Let Us Java", 1st Edition, PBP Publications, 2012 .
4. C. Thomas Wu, "An Introduction to Object-Oriented programming with Java", 5th Edition Tata McGraw-Hill Publishing company Ltd 2010.

Reference Books:

R1->Cay S. Horstmann and Gary Cornel, "Core Java: Volume I – Fundamentals", 8th Edition,
Sun Microsystems Press, 2011

R2-> Timothy Budd "Understanding Object-oriented programming with Java" Pearson Education, 2nd edition, 2006

R3-> Yashawabt Kanetkar, "Let Us Java ", 1 st Edition, BPB Publications 2012

Web pages:

1. http://www.javatpoint.com
2. http://www.javatutorialpoints
3. www.w3schools.com/

Staff-incharge **HOD / CSE**

UNIT I **OVERVIEW**

Why Object-Oriented Programming in C++ – Native Types and Statements –Functions and Pointers- Implementing ADTs in the Base Language.

1.Object–Oriented Programming Concepts:

The important concept of OOPs are:

Objects

Classes

Inheritance

Data Abstraction

Data Encapsulation

Polymorphism

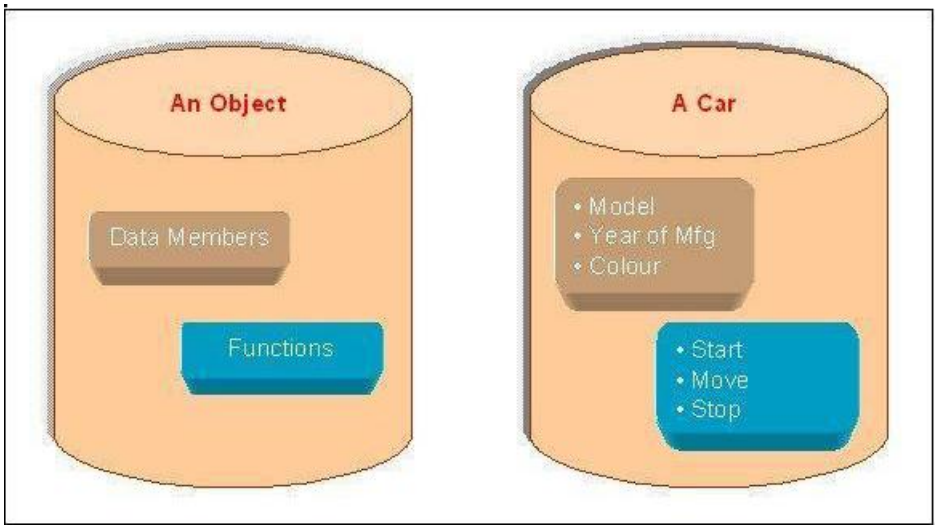
Overloading

Reusability

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one

instance of an object. Each instance of an object can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represent a set of individual

objects. Characteristics of an object are represented in a class as **Properties**. The actions that can be performed by objects becomes functions of the class and is referred to as **Methods**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR*

represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, Stop form the

Methods of *Car* Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance:

Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class*, The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class.* Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without

presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.
Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language.

Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

```
#include <iostream>
class employee // Class Declaration
{
private:
char empname[50];
int empno;
public:
void getvalue()
{
cout<<"INPUT Employee Name:";
cin>>empname;
cout<<"INPUT Employee Number:";
cin>>empno;



}

void displayvalue()
{
cout<<"Employee Name:"<<empname<<endl;
cout<<"Employee Number:"<<empno<<endl;
}
};

main()
{
employee e1; // Creation of Object
```

```
e1.getvalue();

e1.displayvalue();

}
```

2.Programming Concepts:

Encapsulation

It is a mechanism that associates the code and the data it manipulates into a single unit to and keeps them safe from external interference and misuse. In C++ this is supported by construct called class. An instance of an object is known as object which represents a real world entity.

Data Abstraction

A data abstraction is a simplified view of an object that includes only features one is interested in while hides away the unnecessary details. In programming languages, a data

abstraction becomes an abstract data type or a user-defined type. In OOP, it is implemented as a class.

Inheritance:

Inheritance is a means of specifying hierarchical relationships between types C++ classes can inherit both data and function members from other (parent) classes. Terminology: "the child (or derived) class inherits (or is derived from) the parent (or base) class".

Polymorphism:

Polymorphism is in short the ability to call different functions by just using one type of

function call. It is a lot useful since it can group classes and their functions together. Polymorphism means that the same thing can exist in two forms. This is an important characteristic of true object oriented design - which means that one could develop good OO design with data abstraction and inheritance, but the real power of object oriented design

seems to surface when polymorphism is used.

Multiple Inheritance

The mechanism by which a class is derived from more than one base class is known as multiple inheritance. Instances of classes with multiple inheritance have instance variables

for each of the inherited base classes.

Basic c++:

A class definition begins with the keyword class.

The body of the class is contained within a set of braces, { } ; (notice the semi-*colon).* class class_name

{

….

….

….

};

Within the body, the keywords *private:* and *public:* specify the access level of the members of the class.

–

the default is private.

Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.

```
class class_name
{
private:
…
…
…
public:

…
…
…
};
```

Example:

This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

```
class Circle
{
private:
double radius;
public:
void setRadius(double r);
double getDiameter();
double getArea();
double getCircumference();
};
```

Member access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

private

public

protected

A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.

Public members are accessible from outside the class. .

A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

Data Members :

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types.

You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a const static data member of integral or enumeration type may have an explicit initializer.

A class X cannot have a member that is of type X, but it can contain pointers to X, references to X, and static objects of X. Member functions of X can take arguments of type X and have a return type of X. For example:

```
class X

{

X();

X      *xptr;        X
&xref; static X
xcount;              X
xfunc(X);
```

};

Static members:

Class members can be declared using the storage class specifier static in the class member list. Only one copy of the static member is shared by all objects of a class in a program.

When you declare an object of a class having a static member, the static member is not part of the class object.

You access a static member by qualifying the class name using the :: (scope resolution) operator. In the following example, you can refer to the static member f() of class type X as X::f() even if no object of type X is ever declared:

```
struct X {

static int f();

};


int main() {

X::f();

}
```

Function:

Functions are building blocks of the programs. They make the programs more modular and easy to read and manage. All C++ programs must contain the function main( ). The execution of the program starts from the function main( ). A C++ program can contain any number of functions according to the needs. The general form of the function is: -

```
return_type function_name(parameter list)

{

body of the function

}
```

The function of consists of two parts function header and function body. The function header is:-

return_type function_name(parameter list)

The return_type specifies the type of the data the function returns. The return_type can be void which means function does not return any data type. The function_name is the name of the function. The name of the function should begin with the alphabet or underscore. The

parameter list consists of variables separated with comma along with their data types. The parameter list could be empty which means the function do not contain any parameters. The parameter list should contain both data type and name of the variable. For example,

int factorial(int n, float j)

is the function header of the function factorial. The return type is of integer which means function should return data of type integer. The parameter list contains two variables n and j of type integer and float respectively. The body of the function performs the computations.

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the friend

specifier.

These are called friends of a class. You can declare a member function as static; this is called a static member function. A member function that is not declared as static is called a

nonstatic member function.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the

declaration of that member in the class member list. When the function add() is called in the following example, the data variables a, b, and c can be used in the body of add().

```
class x
{
public:
int add() // inline member function add
{return a+b+c;};
private:
int a,b,c;
};
```

Special type of member function:

Constructor:

–

Public function member

–

called when a new object is created (instantiated).

–

Initialize data members.

–

Same name as class

–

No return type

–

Several constructors

Function overloading

```
class Circle
{
private:
double radius;
public:
Circle();
Circle(int r);
void setRadius(double r);
double getDiameter();
double getArea();
double getCircumference();
};
```

Default arguments:

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

long int sum(int n,int diff=1,int first_term=1 )
```

```
{
long sum=0;;
for(int i=0;i<n;i++)
{
cout<<setw(5)<<first_term+ diff*i;
sum+=first_term+diff*i;
}
return sum;
}

int main()
{

cout<<endl<<Sum=<<setw(7)< <sum(10)<<endl;
//first term=1; diff=1,n=10

//sums the series 1,2,3,4,5……10


cout<<endl<<Sum=<<setw(7)< <sum(6,3,2)<<endl;
//first term=1; diff=2,n=10 //sums the series
2,5,8,11,14,17


cout<<endl<<Sum=<<setw(7)< <sum(10,2)<<endl;
//first term=1; diff=2,n=10

//sums the series 1,3,5……..19


return 1;
}
```

all the parameters with default values should lie to the right in the signature list i.e. the default arguments should be the trailing arguments—those at the end of the list.

when a function with default arguments is called, the first argument in the call statement is assigned to the first argument in the definition, the 2nd to 2nd and so on.

This becomes more clear from the last call to sum() in the above example where value 10 is assigned to n and 2 is assigned to diff and not first_term.

the default argument values appear in the prototype as well as definition.

You still may omit variable names in the prototypes.

The syntax then being

int xyz(int =2,char=5);

Function Overloading:

*C++ supports writing more than one function with the same name but different argument lists. This could include:

–

different data types

_

different number of arguments

*The advantage is that the same apparent function can be called to perform similar but different tasks. The following will show an example of this **.** void **swap** (int ***a,*** int ***b***) ;

void **swap** (float ***c,*** float ***d***) ;

void **swap** (char ***p,*** char ***q***) ;

int main ( )

{

int **a = 4, b = 6** ;

float **c = 16.7, d = -7.89** ;

char **p = 'M' , q = 'n'** ;

swap (**&a, &b**) ;

swap (**&c, &d**) ;

swap (**&p, &q**) ;

}

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

{

int **temp**; **temp = *a**; ***a = *b**; ***b = temp**; }

void **swap** (float ***c,*** float ***d***)

{

float **temp**; **temp = *c**; ***c = *d**; ***d = temp**;

}

void **swap** (char ***p,** char ***q**)

{

char **temp**; **temp = *p**; ***p = *q**; ***q = temp**;

}

Friend Function:

*A friend function of a class is defined outside the class‘s scope (I.e. not member functions), yet has the right to access the non-public members of the class.

*Single functions or entire classes may be declared as friends of a class.

*These are commonly used in operator overloading. Perhaps the most common use of friend

functions is overloading << and >> for I/O.

*Basically, when you declare something as a friend, you give it access to your private data members.

*This is useful for a lot of things – for very interrelated classes, it more efficient (faster) than using tons of get/set member function calls, and they increase **encapsulation** by allowing

more freedom is design options.

*A class doesn't control the scope of friend functions so friend function declarations are usually written at the beginning of a .h file. Public and private don't apply to them.

Example pgm//

```
class someClass

{

friend void setX( someClass&, int);

int someNumber;

...     rest of class definition
}


//    a function called setX defined in a program
void setX( someClass &c, int val) {


c.someNumber = val; }


//inside a main function
someClass myClass;
setX (myClass, 5);

//this will work, since we declared

//setX as a friend
```

Const Functions :

If you declare a class method const, you are promising that the method won't change the value of any of the members of the class. To declare a class method constant, put the keyword const after the parentheses but before the semicolon. The declaration of the

constant member functionSomeFunction() takes no arguments and returns void. It looks like this:

```
void SomeFunction() const;
```

Access or functions are often declared as constant functions by using the const modifier Declare member functions to be const whenever they should not change the object

Volatile Functions:

The volatile keyword is a type qualifier used to declare that an object can be modified in the program by something such as the operating system, the hardware, or a concurrently

executing thread. If your objects are used in a multithreaded environment or they can be accessed asynchronously (say by a signal handler), they should be declared volatile. A volatile object can call only volatile member functions safely. If the program calls a

member function that isn't volatile, its behavior is undefined. Most compilers issue a warning if a non-volatile member function is called by a volatile object:

```
struct S

{int f1();

int f2() volatile;

}
```

Static Members:

Object Oriented Programming in C++. A class member is either a property or a method. A static member of a class is a member whose value is the same for every object instantiated.

This means that if one object changes the value of the static member, this change will be

reflected in another object instantiated from the class. The change (or the resulting value) will be the same in all the instantiated objects. You can also access a static member using the class name without instantiation. In this part of the series, we look at static members in C++

classes. You can have a static member along side other members in your class.

Static Property

A static property is also called a static data member.

Declaring a Static Property

You declare a static property just as you declare any other attribute, but you precede the declaration expression with the keyword, static and a space. The syntax is:

static Type Ident;

Despite this simple feature, you have to learn how to use the static member. You do not use it in the straightforward way.

Example

The following class illustrates the use of a static property member:

```
#include <iostream>

using namespace std;

class MyClass

{

public:

static int sameAll;

};

int MyClass::sameAll = 5;

int main()

{

MyClass myObj;

myObj.sameAll = 6;
```

```
cout << MyClass::sameAll;


return 0;

}
```

In the code, you have a class called MyClass. This class has just one member, which is the static data member. You initialize the static member outside the class description as shown above. You begin with the return type of the static property. This is followed by a space and then the name of the class. After that you have the scope operator, then the identifier of the static property. Then you have the assignment operator and the value.

You instantiate an object form the class that has the static member in the normal way. Line

1

in the main function illustrates this. You access the static property of an instantiated object in the normal way. The second line in the main function illustrates this. However, changing the value as this line has done means changing the value for the class (description) and any

instantiated object and any object that is still to be instantiated.

The third line in the main function displays the static property value. It uses the class name; it did not use the object name. To use the class name to access the static attribute, you begin with the class name. This is followed by the scope operator and then the identifier of the static property. This shows how you can access a static attribute with the class name directly and without using an object; this is like accessing the property in the class description. The static member is a kind of global object.

Objects:

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an *object*. A *class*

is an extended concept similar to that of *structure* in C programming language, this class describes the data properties alone. In C++ programming language, *class* describes both the properties (data) and behaviors (functions) of objects. *Classes* are not *objects*, but they are used to instantiate *objects*.

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Juts as we declare a variable of data type *int* as:

int x;

Objects are also declared as:

class name followed by object name;

exforsys e1;

This declares *e1* to be an object of class exforsys.

For example a complete class and object declaration is given below:

class exforsys

```
{
private:
int x,y;
public:
void sum()
{
……
……
}

};


main()
{
exforsys e1;
……
……
}
```

The object can also be declared immediately after the class definition. In other words the

object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

Pointers and Objects:

```
#include <iostream>
using namespace std;
class myclass {
int i;
public:
myclass(int j) {
i = j;
}

int getInt() {
return i;
}
};
int main()
{
myclass ob(88), *objectPointer;
objectPointer = &ob; // get address of ob
cout << objectPointer->getInt(); // use -> to call getInt()
return 0;
}
```

Constant objects:

We've already seen const references demonstrated, and they're pretty natural: when you declare a const reference, you're only making the data referred to const. References, by their very nature, cannot change what they refer to. Pointers, on the other hand, have two ways that you can use them: you can change the data pointed to, or change the pointer itself.

Consequently, there are two ways of declaring a const pointer: one that prevents you from changing what is pointed to, and one that prevents you from changing the data pointed to.

The syntax for declaring a pointer to constant data is natural enough:

```
const int *p_int;
```

You can think of this as reading that *p_int is a "const int". So the pointer may be changeable, but you definitely can't touch what p_int points to. The key here is that the const appears before the *.

On the other hand, if you just want the address stored in the pointer itself to be const, then you have to put const after the *:

```
int x;

int * const p_int = &x;
```

Personally, I find this syntax kind of ugly; but there's not any other obviously better way to do it. The way to think about it is that "* const p_int" is a regular integer, and that the value stored in p_int itself cannot change--so you just can't change the address pointed to. Notice, by the way, that this pointer had to be initialized when it was declared: since the pointer itself is const, we can't change what it points to later on! Them's the rules.

Generally, the first type of pointer, where the data is immutable, is what I'll refer to as a "const pointer" (in part because it's the kind that comes up more often, so we should have a

natural way of describing it).

Nested Classes:

§

A class can be declared within the scope of another class. Such a class is called a

"nested class." Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope. To refer to a nested class from a scope other than its immediate enclosing scope, you must use a fully qualified name value class Outside { value class Inside { }; }; In the same way, you can nest as many classes as you wish in another class and you can nest as many classes inside of other nested classes if you judge it necessary.

Just as you would manage any other class so can you exercise control on a nested class. For example, you can declare all necessary variables or methods in the nested class or in the nesting class. When you create one class inside of another, there is no special programmatic relationship between both classes: just because a class is nested doesn't mean that the nested class has immediate access to the members of the nesting class. They are two different classes and they can be used separately.

§

The name of a nested class is not "visible" outside of the nesting class. To access a nested class outside of the nesting class, you must qualify the name of the nested class

anywhere you want to use it. This is done using the :: operator. For example, if you want to declare an Inside variable somewhere in the program but outside of Outside, you must qualify its name. Here is an example:

Local classes:

A local class is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

int x; // global variable

void f() // function definition

```
{
static int y; // static variable y can be used by
// local class
int x; // auto variable x cannot be used by
// local class

extern int g(); // extern function g can be used by
// local class


class local // local class
{
int g() { return x; } // error, local variable x
// cannot be used by g
int h() { return y; } // valid,static variable y
int k() { return ::x; } // valid, global x
int l() { return g(); } // valid, extern function g
};
}
```

```
int main()

{

local* z; // error: the class local is not visible

// ...}
```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword inline.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()

{

class local


{




int f(); // error, local class has noninline

// member function


int g() {return 0;} // valid, inline member
function static int a; // error, static is not allowed
for // local class


int b; // valid, nonstatic variable

};

}

//
```

UNIT II BASIC CHARACTERISTICS OF OOP 9

Data Hiding and Member Functions- Object Creation and Destruction- Polymorphism data abstraction:

Iterators and Containers.

A Real Programming Example

we want to program a card deck for a simple blackjack game we are programming. Remember containership and inheritance? Let's think about the types of parts that make up

a deck -- and those are the cards. Since all of the cards are very similar in structure, we could use a struct to represent a single card:

```
enum Suit = {Clubs, Spades, Diamonds, Hearts};

struct Card {

Suit suit;

char digit;

};
```

A note on the digit. If digit <= 10, then it is a number, else it is the letter of the card (J, Q, K, A). You could also use it as a number 1(ace) through 13(king) as well, perhaps if you were using the card value in additions or such.

A simple object, the card deck only has one type of item. Now let's think about the types of actions you can perform on the deck, and then make a class declaration out of this list, as well as using the previously declared data.

```
class Deck {

public:

void CreateDeck();//Fills array with legal cards

void Shuffle(); //Shuffles those cards

Card DrawCard(); //Gets a card from the deck

private:

Card cards[52];

};
```

Now we have considered all of the things we may need to use a card deck. The programmer first sets up the deck with CreateDeck(), then whenever needed can Shuffle() the deck, and when the dealer deals a card, it can be picked up using DrawCard(), and then perhaps placed in the

players hand (which could also could be a class too) or whatever the programmer needs to do with it.

Constructors and Destructors

In object-oriented programming, a **constructor** (sometimes shortened to **ctor**) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables

required when the object is first created.

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The

syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()

In the above example the arguments is optional.

The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.

I'm not going to write all of the code for this class. Instead I'll leave it open as an exercise to practice on working with classes. But let's focus on the CreateDeck() function for now. You may have already noticed that variables aren't being properly pre-initalized. In C this was easy since all variables were public, and you could simply zero them out, but in classes, some of the members are private. Now what? The programmer could call the CreateDeck()

function, but even so, if the programmer forgets to do this the rest of the functions could crash the program. C++'s solution to this is called the constructor. A constructor is the function which allocates memory for the object as well as initalizing it. Every variable in C++ has a constructor, even the basic types, but the compiler takes care of these issues for you. However with classes, even though the compiler can allocate memory for you, it will not initalize them, so the remain undefined, as with any other variable. Also, when the variable goes out of scope, the memory needs to be deleted, requireing the constructors counterpart, the destructor.

A constructor is declared by creating a function by the same name as the class. The destructor has the same name, except with a ~ (the tlide key, next to the 1) in front of it. Below is the modified Deck class which takes advantage of C++ constructors and destructors. The array

change to a pointer is to allow for dynamic memory allocation using the new command, to show a very common use of the constructor and destructor: class Deck {

public:

Deck(); //Constructor

~Deck(); //Destructor

void CreateDeck();//Fills array with legal cards

void Shuffle(); //Shuffles those cards

Card DrawCard(); //Gets a card from the deck

private:

Card* cards;

```
};
```

Notice that obviously the constructor does not return anything, since it is an "invisible function" which is automatically called when you make the statement Deck MyCardDeck;.

The same is true for the destructor, which is called when the variable goes out of scope. Below is an example of how to define and code a constructor, where the array is allocated and the data is initalized, and the destructor compliment, which cleans up what the constructor did:

```
Deck::Deck() {

cards = new Card[52];//Allocate memory

CreateDeck(); //Set up the deck

}



Deck::~Deck() {

delete[] cards; //Deallocate memory
```

}

Some of the differences between constructors and other Java methods:

Constructors never have an explicit return type.

Constructors cannot be directly invoked (the keyword ―new‖ must be used).

Constructors cannot be synchronized, final, abstract, native, or static.

Constructors are always executed by the same thread.

Some important points about constructors:

A constructor takes the same name as the class name.

The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.

No return type is specified for a constructor.

The constructor must be defined in the public. The constructor must be a public member.

Overloading of constructors is possible. This will be explained in later sections of this tutorial.

How do you differentiate between a constructor and normal function?

Latest Answer : A constructor is a member function of a class that is used to create objects of that class. It has the same name as the class itself, has no return type, and is invoked using the new operator. An ordinary member function has its own name, a return type ...

Default Constructor

Default Constructor

The deafual constructor is a constructor. It may be contains arguemnets (default arquemnets) or It may not be contains arquemnet

Example

```
class A

{

public:

A(){/*body*/} //Deafult constructor without arquemnet

or

A(int a 0 int b 0){/*body*/} //Default constructor with default
arquement };
```

if you did not write any constructor within class A. The implicit constructor or inline constructor A::A() /*[without no body] */ will be called when you create object for class A

Note: you can't use both constructor in same class. Confilict occur when you create object of class A that whether to call first one or second one (ambiguity)

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For

class Exforsys

example:

```
{

private:

int a,b;

public:

Exforsys();

...

};
```

```
Exforsys :: Exforsys()

{

a=0;

b=0;

}
```

In C++, default constructors are significant because they are automatically invoked in certain circumstances:

When an object value is declared with no argument list, e.g. MyClass x;; or allocated dynamically with no argument list, e.g. new MyClass; the default constructor is used to initialize the object

When an array of objects is declared, e.g. MyClass x[10];; or allocated dynamically, e.g. new MyClass [10]; the default constructor is used to initialize all the elements

When a derived class constructor does not explicitly call the base class constructor in its initializer list, the default constructor for the base class is called

When a class constructor does not explicitly call the constructor of one of its object-valued fields in its initializer list, the default constructor for the field's class is called In the

standard library, certain containers "fill in" values using the default constructor when the value is not given explicitly, e.g. vector<MyClass>(10); initializes the vector with 10

elements, which are filled with the default-constructed value of our type.

In the above circumstances, it is an error if the class does not have a default constructor. The compiler will implicitly define a default constructor if no constructors are explicitly

defined for a class. This implicitly-declared default constructor is equivalent to a default constructor defined with a blank body.

(Note: if some constructors are defined, but they are all non-default, the compiler will not implicitly define a default constructor. This means that a default constructor may not exist for a class.).

Parameterized Constructor – Class Interface

These are access specifiers for class data members and member methods.

1.

Public: The data members and methods having public as access specifier can be accessed by the class objects created outside the class.

2.

Protected: The data members and methods declared as protected will be accessible to the class methods and the derived class methods only.

3.

Private: These data members and methods will be accessible from the class methods only not from derived classes and not from objects created outside the class.

4.

Internal: Some languages define internal as an access specifier which means the data member or method is available to all the classes inside that particular assembly.

5.

Friend: A friend class or method can access all data of a class including private and protected data.

// A parameterized constructor.

```
using System;

class MyClass {
public int x;

public MyClass(int i) {
x = i;
```

```
}
}

public class ParmConsDemo {
public static void Main() {
MyClass t1 = new MyClass(10);
MyClass t2 = new MyClass(88);

Console.WriteLine(t1.x + " " + t2.x);
}
}
```

Constructor with Dynamic Allocation

Dynamic memory allocation (also known as heap-based memory allocation) is the allocation

of memory storage for use in a computer program during the runtime of that program.

A process of obtaining access to additional memory during program execution.

```
void YourClass::deleteAll()
{
delete ptr1; ptr1 = 0;
delete ptr2; ptr2 = 0;
delete ptr3; ptr3 = 0;
```

```
}

YourClass::YourClass():
ptr1(0), ptr2(0), ptr3(0)
{
try
{
ptr1 = new whatever;
ptr2 = new whatever;
ptr3 = new whatever;

}
catch(...)
{
deleteAll();
}
}

YourClass::~YourClass()
{
deleteAll();
}
```

What is static memory allocation and dynamic memory allocation?

Static Memory Allocation: Memory is allocated for the declared variable by the compiler. The address can be obtained by using _address of‟ operator and can be assigned to a pointer.

The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory allocation.

Dynamic Memory Allocation: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions

when value is returned by functions and assigned to pointer variables.

Copy Constructor

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

For example to invoke a copy constructor the programmer writes:

Exforsys e3(e2);

or

Exforsys e3=e2;

Copy constructor is

1. a constructor function with the same name as the class
2. used to make deep copy of objects.

There are 3 important places where a copy constructor is called.

1. When an object is created from another object of the same type
2. When an object is passed by value as a parameter to a function
3. When an object is returned from a function.

If a copy constructor is not defined in a class the compiler itself defines one. This will ensure a shallow copy. If the class does not have pointer variables with dynamically

allocated memory then one need not worry about defining a copy constructor. It can be left to the compiler's discretion.

For Example:

```
#include <iostream.h>

class Exforsys()
```

```
{
private:
int a;
public:
Exforsys()

{ } Exforsys(int
w)
{
a=w;
}
Exforsys(Exforsys& e)
{
a=e.a;
cout<<‖ Example of Copy Constructor‖;
}
void result()
{
cout<< a;
}
};



void main()
{
Exforsys e1(50);
Exforsys e3(e1);
cout<< —=‖;e3.result();
```

}

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

Example of Copy Constructor

e3=50

difference between copy constructor and constructor

Constructor is called when an object is created.Copy constructor is called when the copy of an object is made. For e.g. passing parameter to function by value function returning by value. Copy constructor takes the parameter as const reference to the object.

A copy constructor is called whenever an object is passed by value, returned by value or explicitly copied.

Destructors

The destructor of an automatic object is called when the object goes out of scope. The destructor itself does not actually destroy the object, but it does perform termination housekeeping before the system reclaims the object‘s.

What is the use of Destructors

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

Destructors take the same name as the class name.

Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.

The Destructor does not take any argument which means that destructors cannot be overloaded.

No return type is specified for destructors.

For example:

```
class Exforsys

{

private:

……

public:

Exforsys()

{ }


~ Exforsys()
{ }
}
```

What is defference between constructor and destructor

Like constructor the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example the destructor of a class integer can be define as :-

~integer(){}

A destructor never takes any argument nor does it return any value. It will invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

Re: What is role of constructor and destructor in C++?

The constructor's job is to set up the object so that it can be used.Destructors are less complicated than constructors. You don't call them explicitly (they are called automatically for you), and there's only one destructor for each object. The name of the destructor is the name of the class, preceeded by a tilde.

Operator Overloading

Operator Overloading in two Parts, In Part I of Operator Overloading you will learn about Unary Operators, Binary Operators and Operator Overloading – Unary operators.

Operator overloading is a very important feature of Object Oriented Programming. Curious to know why!!? It is because by using this facility programmer would be able to create new

definitions to existing operators. In fact in other words a single operator can take up several functions as desired by programmers depending on the argument taken by the operator by

using the operator overloading facility.

After knowing about the feature of operator overloading now let us see how to define and use this concept of operator overloading in C++ programming language.

We have seen in previous sections the different types of operators. Broadly classifying operators are of two types namely:

Unary Operators

Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely ++

called as Increment operator, -- called as Decrement Operator, ! , ~, unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators all this which we have seen in previous section of operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword operator.

The general syntax for defining an operator overloading is as follows:

return_type classname :: operator operator symbol(argument)

{

……..

statements;

}

Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

return_type – is the data type returned by the function

class name - is the name of the class

operator – is the keyword

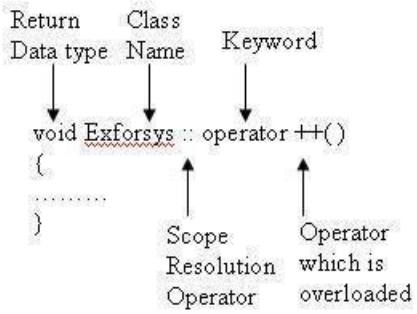
operator symbol – is the symbol of the operator which is being overloaded or

defined for new functionality

:: - is the scope resolution operator which is used to use the function definition outside the class. The usage of this is clearly defined in our earlier section of How to define class members.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

Class Exforsys

{

Private :

4. … … .
Public :

void operator++();

… … … .
};

So the important steps involved in defining an operator overloading in case of unary operators are namely:

Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function. The concept of friend function we will define in later sections. If in this case of unary operator overloading if the function is a member function then the number of arguments taken by the operator member function is none as

seen in the below example. In case if the function defined for the operator overloading is a friend function which we will discuss in later section then it takes one argument.

The operator overloading is defined as member function outside the class using the scope resolution operator with the keyword operator as explained above

Now let us see how to use this overloaded operator member function in the program

```
#include<iostream.h>

classExforsys

{

private:

int

x;


public:

Exforsys( )

{x=0;


}
//Constructor
void display();


void

Exforsys++();

};
```

```
Void Exforsys::display()
{
cout<<‖
of
x
is:—<<x;
}

void Exforsys::operator ++( )//Operator Overloading for operator
++ defined
{

++x;
}


Void main()
{
Exforsys e1,e2; //Object e1 and e2 created
cout<<‖Before
Increment‖
cout<<‖:‖<<e1.display();
cout<<‖
e2:‖<<e2.display();
++e1; //Operator

overloading applied
++e2;

cout<<‖
After Increment‖
```

```
cout<<‖
e1:‖<<e1.display();
cout
<<‖e2:
‖<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of
x
is:0
Object e1:
Value of
x
is:0
Before Increment
Object e1:
Value of
x
is:1
```

Object e1:

Value of x is: 1

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero.

When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are

displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

This is how unary operators get overloaded. We will see in detail how to overload binary operators in next section.

Overloading through Friend Functions

Need for Friend Function:

when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access

private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them

friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++:

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

It is possible to declare a function as friend in any number of classes.

When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.

A friend function, even though it is not a member function, would have the rights to access the private members of the class.

It is possible to declare the friend function as either private or public.

The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example to understand the friend function:

```
#include

class exforsys

{

private:

int a,b;

public:

void test()

{

a=100;

b=200;

}

friend int compute(exforsys e1)


//Friend Function Declaration with keyword friend and with the object of
class exforsys to which it is friend passed to it };



int compute(exforsys e1)


{

//Friend Function Definition which has access to private data

return int(e1.a+e2.b)-5;

}


main()

{

exforsys e;
```

```
e.test();

cout<<"The result is:"<<COMPUTE(E);

//Calling of Friend Function with object as argument.

}
```

The output of the above program is

```
The result is:295
```

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as a friend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

```
friend int compute (exforsys e1)
```

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class exforsys. It is declared as friend inside the class, the private data values a and b are added, 5

is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown above.

Friend Function Overload

The concept operator overloading and friend function are supported by Java by defaault only + operator is overloaded over string .

For example ---

String abc ;

String s1 Hello ;

String s2 Geek Interview ;

abc s1+s2;

Sop(abc);

would print Hello Geek Interview

Overloading the Assignment Operator

The operators available in C++ programming language are:

Assignment Operator denoted by =

Arithmetic operators denoted by +, -, *, /, %

Compound assignment Operators denoted by +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

Increment and Decrement operator denoted by ++, --

Relational and equality operators denoted by ==, !=, >, <, >=, <=

Logical operators denoted by !, &&, ||

Conditional operator denoted by ?

Comma operator denoted by ,

Bitwise Operators denoted by &, |, ^, ~, <<, >>

Explicit type casting operator

sizeof()

Assignment Operator

This is denoted by symbol =. This operator is used for assigning a value to a variable. The left of the assignation operator is known as the lvalue (left value), which must be a variable. The right of the assignation operator is known as the rvalue (right value). The rvalue can be a constant, a variable, the result of an operation or any combination of these.

For example:

x = 5;

By following the right to left rule the value 5 is assigned to the variable x in the above assignment statement.

Arithmetic operators

The operators used for arithmetic operation sin C++ are:

+ For addition

- For subtraction

* For multiplication

/ For division

% For modulo

Compound assignment Operators

This operator is used when a programmer wants to update a current value by performing operation on the current value of the variable.

For example:

Old

+=

new

is

equal to

Old = old + new

Compound assignment operators function in a similar way the other operators +=, -=, *=, /=,

%=, >>=, <<=, &=, ^=, |= function.

Increment and Decrement Operator

The increment operator is denoted by ++ and the decrement operator by --. The function of the increment operator is to increase the value and the decrement operator is to decrease the value. These operators may be used as either prefix or postfix. A Prefix operator is written before the variable as ++a or –a. A Postfix operator is written after the variable as a++ or a--.

The Functionality of Prefix and Postfix Operators

In the case that the increment or decrement operator is used as a prefix (++a or –a), then the value is respectively increased or decreased before the result of the expression is evaluated.

Therefore, the increased or decreased value, respectively, is considered in the outer expression. In the case that the increment or decrement operator is used as a postfix (a++ or a--), then the value stored in a is respectively increased or decreased after being evaluated.

Therefore, the value stored before the increase or decrease operation is evaluated in the outer expression.

For Example:

y=3;

x=++y; //Prefix : Here Value of x becomes 4

But for the postfix operator namely as below:

y=3 //Postfix : Here Value of x is 3 and Value of y is 4
x=y++;

Relational and Equality Operators

These operators are used for evaluating a comparison between two expressions. The value returned by the relational operation is a Boolean value (true or false value). The operators used for this purpose in C++ are:

== Equal to

!= Not equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

we can overload assignment operator as a normal...

If the operation modifies the state of the class object it operates on it must be a member function not a friend function Thus all operator such as * + etc are naturally defined as member functions not friend functions Conversely if the operator does not modify any of its operands but needs only a representation of the object it does not

have to be a member function and often less confusing. This is the reason why binary operators are often implemented as friend functions such as + * - etc..

Why friend function cannot be used to overload the assignment operator?

A friend function is a non-member function of the class to which it has been

defined as friend.Therefore it just uses the functionality(or the functions and data) of the class so. it wont consists the inplementation for that class.(any)

we can overload assignment operator as a normal function.But we can not overload assignment operator as friend function why?

Because assignment operator is one of the default method provided by the class.

Type Conversion

What is Type Conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

There are two ways of achieving the type conversion namely:

Automatic Conversion otherwise called as **Implicit Conversion**

Type casting otherwise

called as **Explicit Conversion**

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words value gets automatically converted to the specific type in which it is assigned.

Let us see this with an example:

```
#include
<iostream.h>
void
main()
{
Short x=6000;
int
y;
y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as

In C++ the type casting can be done in either of the two ways mentioned below namely:

C-style casting

C++-style casting

The C-style casting takes the synatx as

(type) expression

This can also be used in C++.

Apart from the above the other form of type casting that can be used specifically in C++ programming language namely C++-style casting is as below namely: type (expression)

This approach was adopted since it provided more clarity to the C++ programmers rather

than

the

C-style casting.

Say for instance the as per C-style casting

(type) firstVariable * secondVariable

is not clear but when a programmer uses the C++ style casting it is much more clearer as below

type (firstVariable) * secondVariable

Let us see the concept of type casting in C++ with a small example:

#include

<iostream.h>

void

main()

{

int

a;

float

b,c;

cout<< —Enter the value of a:‖;

cin>>a;

cout<< —n Enter the value of b:‖;

cin>>b;

c=float(a)+b;

cout<<‖n

The

value of

```
c
is:‖<<c;
}
```

The output of the above program is

```
Enter the
value of
a:
10
Enter the
value of
b:
12.5
The value of c is: 22.5
```

In the above program a is declared as integer and b and c are declared as float. In the type conversion statement namely

```
c = float(a)+b;
```

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable

c with value as 22.5

Explicit Constructor

Explicit Constructor

Explicit constructor is actually a parameterized constructor which takes some parameters in order to create instance of a class.

E.g. Class Sample

Sample [a b]

The explicit keyword in C++ is used to declare explicit constructors. Explicit constructors are simply constructors that cannot take part in an implicit conversion. Consider the following example:

```
class Array
{
public:
Array(size_t count);
//etc.
};
```

explicit B (const A& aObj)

But explicit on a constructor with multiple arguments has no effect, since such constructors cannot take part in implicit conversions. However, explicit will have an effect if a constructor has multiple arguments and all but one of the arguments has a default value.

For Example:

The code

Code: Cpp

```
#include<iostream.h>

class A
{
int data1;
int data2;

char* name;

public:
A(int a, int b=10, char* c = "mridula"):data1(a), data2(b), name(c)
{
cout<<"A::Construcor... ";
};

friend void display(A obj);
};
```

```
void display(A obj)

{


cout<<"Valud of data1 in obj := "<< obj.data1<<endl;
cout<<"Valud of data2 in obj := "<< obj.data2<<endl;
cout<<"Valud of name in obj := "<< obj.name<<endl;
}


int main()

{

//Call display with A object i.e. a1

display(100);


return (0);


}
```

Output:

```
----------

./a.out

A::Construcor...

Valud of data1 in obj := 100

Valud of data2 in obj := 10

Valud of name in obj := mridula
```

In this example, though we have multiple argument constructor

A(int a, int b=10, char* c = "mridula") (all defaulted argument but one), but still there seems to be an implicit conversion happening from type int to A's object.

As said above, it is better to use **explicit** for such constructor declaration too to avoid any such implicit conversions.

Declare the multiple argumented constructor in the above example as below to avoid implicit conversion:

explicit A(int a, int b=10, char* c = "mridula");

Part-A (2 - Marks)

1. What is a Constructor?
2. Define Copy constructor.
3. Define Destructor.
4. What is Operator Overloading?
5. Define default constructor.
6. What is the difference between Explicit and Implicit casting?
7. What is use of Assignment Operator?
8. What is the use of Constructor with Dynamic allocation?
9. Define Explicit constructor.
10. What is meant by Typecasting?

11. What is meant by virtual destructors?
12. Difference between overloading and overriding?
13. What is function overloading?
14. List out the limitations of function overloading.
15. What is overloaded constructor?

Part-B

1. a. Write a program to overload assignment operator. (8)

b. Write a program to convert int to double using typecasting operator. (8)

2. Write a program to overload arithmetic operators through friend function. (16)

3. Write a program to implement Constructor, Copy constructor and Destructor. (16)

4. a. Write a program to demonstrate the use of Explicit Constructor. (8)
b. Write a program for function overloading. (8)

5. Write a program for overloading new and delete operators. (16)

Templates, Generic Programming, and STL-Inheritance-Exceptions-OOP Using C++.

INHERITANCE:

Let us start by defining inheritnace. A very good website for finding computer science definitions is http://www.whatis.com. The definitions in this article are stolen from that website.

Definition: Inheritance

Inheritance is the concept that when a class of object is defined, any subclass that is defined can inherit the definitions of one or more general classes. This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part. This not only speeds up program development; it also ensures an inherent validity to the defined

subclass object (what works and is consistent about the class will also work for the subclass).

The simple example in C++ is having a class that inherits a data member from its parent

class.

```
class A

{

public:

integer d;

};

class B : public A

{

public:

};
```

The class B in the example does not have any direct data member does it? Yes, it does. It

inherits the data member d from class A. When one class inherits from another, it

acquires all of its methods and data. We can then instantiate an object of class B and call

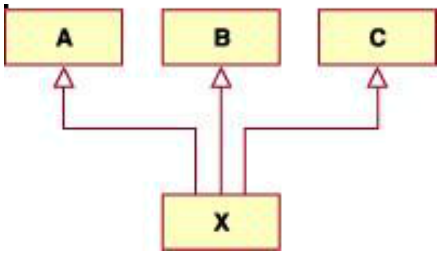
into that data member.

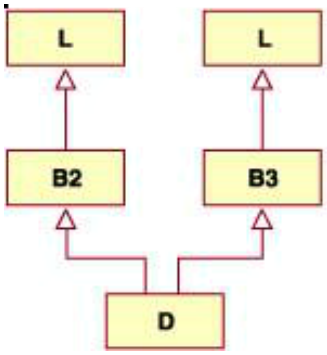
```
void func()

{

B b;

b.d = 10;

};
```

TYPES OF INHERITANCE:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchial Inheritance

1. Single Inheritance - One base class and one derived class.





2. Multiple Inheritance - A class is derived from more than one base classes
3. Multilevel Inheritance - A sub class inherits from a class which inherits from another class.

4. Hierarchical Inheritance - More than one subclass inherited from a single base class.

Multiple inheritance (C++ only)

You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called *multiple inheritance*.

In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };

class B { /* ... */ };

class C { /* ... */ };

class X : public A, private B, public C { /* ... */ };
```

The following *inheritance graph* describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ }; // direct base class

class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown

in the following example:

```
class L { /* ... */ }; // indirect base class

class B2 : public L { /* ... */ };

class B3 : public L { /* ... */ };

class D : public B2, public B3 { /* ... */ }; // valid
```

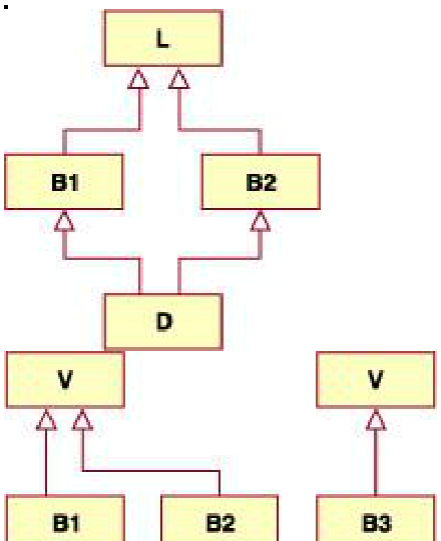
In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

```
B2::L
```

or

```
B3::L.
```

You can also avoid this ambiguity by using the base specifier virtual to declare a base class, as described in Derivation (C++ only).



VIRTUAL BASE CLASSES (C++ ONLY)

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L,

one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

For example:

```
class L { /* ... */ }; // indirect base class

class B1 : virtual public L { /* ... */ };

class B2 : virtual public L { /* ... */ };

class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:

```
class V { /* ... */ };

class B1 : virtual public V { /* ... */ };

class B2 : virtual public V { /* ... */ };

class B3 : public V { /* ... */ };

class X : public B1, public B2, public B3 { /* ... */
};
```

In the above example, class X has two subobjects of class V, one that is shared by classes

B1

and B2 and one through class B3.

VIRTUAL FUNCTIONS:

C++ matches a function call with the correct function definition at compile time known as static binding

the compiler can match a function call with the correct function definition at run time known as dynamic binding.

declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

Example:

```
class A {

public:

virtual void f() { cout << "Class A" << endl; }

};

class B: public A {

public:

void f(int) { cout << "Class B" << endl; }

};

class C: public B {

public:

void f() { cout << "Class C" << endl; }

};
```

—Pure‖ly Virtual: A virtual function declared with no definition

base class contains no implementation at all

class containing a pure virtual function is an **abstract** class
similar to Java interfaces

cannot instantiate from abstract classes

enforces a design through inheritance hierarchy

inherited classes **must** define implementation

Example:

```
class A {
public:
virtual void f() = 0; // pure virtual
};

class B: public A {
public:
void f() { cout << "Class B" << endl; }
};
class C: public B {
public:
void f() { cout << "Class C" << endl; }
};
```

Run Time Type Information (RTTI)

Always exists in OOP: **a prerequisite for dynamic binding** Accessible to programmer?

Not necessarily in statically typed languages

Many things can be done without it!

Almost always in dynamically typed languages

Without it, it is impossible to be sure that an object will recognize a message!

In LST, RTTI is the information accessible from the instance_of pointer

Dynamic binding and casting:

Dynamic Typing: no constraints on the values stored in a variable.

– Usually implies reference semantics

- Run-time type information: dynamic type is

- associated with the value.

  –

  –

  };

  Dynamic casting:

- Casting operator is for polymorphic object casting ,so that it can cast from one object to another object.

- Dynamic cast is also called as safe cast.it succeeds only when the pointer or reference being cast is an object of the target type or derived type from it.

- The syntax is written as dynamic cast<ToobjectptrOr ref>(FromobjectPtrOrRef)

•

If we have a base class and a derived class,casting from derived pointer to base pointer always succeeds.The casting from base pointer to derived can be succeed only if base is actually pointing to an object of derived one. Rtti and templates:

•

If we want to test the type of the actual variable and try to provide validations according to the type we can use RTTI for that.

Cross casting:

It refers to casting from derived to proper base class when there are multiple base classes in case of multiple inheritance.

The dynamic_cast feature of C++ affords another kind of solution -- cross casting.

Consider the following code.

class A {public: virtual ~A();};

class C : public A, public B {};

A* ap = new C;

B* bp = dynamic_cast<B*>(ap);

Notice that classes A and are completely unrelated. Now when we create an

instance of C we can safely upcast it to an . However, we can now take that pointer to A and it to a pointer to a B. This works because the A pointer _ ap ‘ really points at

a C object; and C derives from B Thus, we have cast *accross* the inheritance hierarchy between completely unrelated classes. It should be noted that this will

not work with regular casts since they will not be able to do the address arithmetic to

get the pointer to B correct.

For example:

B* bp = (B*)ap;

While this will compile without errors , it will not generate working code. The value of _ bp ‘ will not actually point to the B part of C. Rather it will still point at the A part of C.

This will lead to undefined behavior

Down casting:

```
rectangle::rectangle(float h, float w, int c, int l):pr(c, l)

{

height = h;


width = w;

xpos = 0;

ypos = 0;

};

 void main()

{

rectangle rc(3.0, 2.0, 1, 3);

C++ statements;

}
```

Part-A (2-Marks)

1. State Inheritance.
2. Write the advantages of multiple Inheritance?
3. Define Polymorphism and also list the types of polymorphism.
4. What are Virtual Functions?
5. What are Virtual members?
6. What is meant by pure virtual function?
7. Give the syntax for pure virtual function?
8. What is meant by RTTI?
9. What is upcasting?
10. What is downcasting?
11. What is cross casting?
12. Give the use of typeid operator?
13. Give the use of dynamic_cast operator?
14. Mention the limitations of RTTI.

Part-B

1. Explain a multilevel, Multiple and Multipath inheritance. (16)
2. Describe about RTTI in detail. (16)
3.Explain

a.Single Inheritance (8)

b.Run time polymorphism (8)

4.Define

Object Oriented Programming 4

SCAD College Of Engineering

(i)Down casting (8)

(ii)Cross casting (8)

5. a. Write a program for pure virtual function. (8)

b. Write a program to implement dynamic casting. (8)

IOSTREAM LIBRARY

-

In Module 5 you have learned the formatted I/O in C by calling various

standard functions. In this Module we will discuss how this formatted I/O implemented in C++ by using member functions and stream manipulators. -

The header files used for formatted I/O in C++ are:

Header file

iostream.h

- Provide basic information required for all stream I/O operation such as cin, cout, cerr and clog correspond to standard input stream, standard output stream, and standard unbuffered and buffered error streams respectively.

iomanip.h

- Contains information useful for performing formatted I/O with parameterized stream

manipulation.

fstream.h

- Contains information for user controlled file processing operations. strstream.h

- Contains information for performing in-memory formatting or in-core formatting. This resembles file processing, but the I/O operation is performed to and from character arrays rather than files.

stdiostrem.h

- Contains information for program that mixes the C and C++ styles of I/O. iostream library

- The compilers that fully comply with the C++ standard that use the template based header files won‘t need the .h extension. Please refer to Module 23 for more information.

- The iostream class hierarchy is shown below. From the base class ios, we have a derived class:

Class

istream

•

Class for stream input operation.

ostream

•

Class for stream output operation.

ios derived classes

-

So, iostream support both stream input and output. The class hierarchy is shown below.

Left and Right Shift Operators

- We have used these operators in most of the Modules in this Tutorial for C++ codes.

- The **left shift operator** (<<) is overloaded to designate stream output and is called **stream insertion operator**.

- The **right shift operator** (>>) is overloaded to designate stream input and is called **stream extraction operator**.

-

These operators used with the **standard stream object** (and with other user defined stream objects) is listed below:

Operators
cin

•

Object of istream class, connected to the **standard input device**, normally the keyboard.

cout

•

Object of ostream class, connected to **standard output device**, normally the display screen.

- For file processing C++ uses (will be discussed in another Module) the following **classes:**

Class

•

ifstream

To perform file input operations.

•

ofstream

For file output operation.

•

fstream

For file input/output operations.

Stream output program example:

```
//string output using <<

#include <stdlib.h>

#include <iostream.h>


void main(void)

{

cout<<"Welcome to C++ I/O module!!!"<<endl;

cout<<"Welcome to ";

cout<<"C++ module 18"<<endl;

//endl is end line stream manipulator

//issue a new line character and flushes the output buffer

//output buffer may be flushed by cout<<flush; command

system("pause");

}
```

get() and getline() Member Functions of Stream
Input - For the get() function, we have three versions.

1. get() without any arguments, input one character from the designated streams including whitespace and returns this character as the value of the function call. It will return EOF when end of file on the stream is encountered. For example:

2. cin.get();

3. get() with a character argument, inputs the next character from the input stream including whitespace. It return false when end of file is encountered while returns a reference to the istream object for which the get member function is being invoked. For example:

4. char ch;
5. cin.get(ch);

7. get() with three arguments, a character array, a size limit and a delimiter (default value _'). It reads characters from the input stream, up to one less than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read.

For example:

```
char namevar[30];
...
cin.get(namevar, 30);
//get up to 29 characters and inserts null
//at the end of the string stored in variable
//namevar. If a delimiter is found,
//the read terminates. The delimiter
//is left in the stream, not stored
```

//in the array.

4. getline() operates like the third get() and insert a null character after the line in the character array. It removes the delimiter from the stream, but does not store it in the character array.

Program examples:

```
//End of file controls depend on system
//Ctrl-z followed by return key - IBM PC
//Ctrl-d - UNIX and MAC


#include <stdlib.h>
#include <iostream.h>
void main(void)

{

char p;


cout <<"Using member functions get(), eof() and put()"
<<"---------------------------------------------"<<endl;
cout<<"Before any input, cin.eof() is "<<cin.eof()<<endl;

cout<<"Enter a line of texts followed by end of file control: "<<endl; while((p = cin.get())

!=EOF)

cout.put(p);

cout<<"some text input, cin.eof() is "<<cin.eof()<<endl; system("pause");

}
```

Program example:

```
//Using read(), write() and gcount() member functions


#include <stdlib.h>

#include <iostream.h>

const int SIZE = 100;

void main(void)

{
```

```
char buffer[SIZE];

cout<<"Enter a line of text:"<<endl;

cin.read(buffer,45);

cout<<"The line of text entered was: "<<endl;

cout.write(buffer, cin.gcount());

//The gcount() member function returns www.tenouk.com

//the number of unformatted characters last extracted

cout<<endl;


system("pause");

}
```

Stream Manipulators

-

Used to perform formatting, such as:

- Setting field width.
- Precision.
- Unsetting format flags.
- Flushing stream.
- Inserting newline in the output stream and flushing the stream.

▪ Inserting the null character in the output stream.
▪ Skipping whitespace.
▪ Setting the fill character in field.

Stream Base

-

For stream base we have:

Operator/function

•

hex

To set the base to hexadecimal, base 16.

•

oct

To set the base to octal, base 8.

•

dec

To reset the stream to decimal.

•

setbase()

Changing the base of the stream, taking one integer argument of 10, 8 or 16 for decimal, base 8 or base 16 respectively. setbase() is parameterized stream manipulator by taking argument, we have to include iomanip.h header file.

Table 18.5: Stream base operator and function.

Program example:

//using hex, oct, dec and setbase stream manipulator

#include <stdlib.h>

#include <iostream.h>

```
#include <iomanip.h>

void main(void)

{

int p;

cout<<"Enter a decimal number:"<<endl;

cin>>p;

cout<<p<< " in hexadecimal is: "

<<hex<<p<<"

<<dec<<p<<" in octal is: "

<<oct<<p<<"

<<setbase(10) <<p<<" in decimal is: "


<<p<<endl;

cout<<endl;

system("pause"); www.tenouk.com

}
```

Floating-point Precision

- Used to control the number of digits to the right of the decimal point.
- Use setprecision() or precision().

- precision 0 restores to the default precision of 6 decimal points.

```
//using precision and setprecision

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

```
#include <iomanip.h>

#include <math.h>

void main(void)

{

double theroot = sqrt(11.55);

cout<<"Square root of 11.55 with various"<<endl;

cout<<" precisions"<<endl;

cout<<"--------------------------------"<<endl;

cout<<"Using 'precision':"<<endl;

for(int poinplace=0; poinplace<=8; poinplace++)

{

cout.precision(poinplace);

cout<<theroot<<endl;


}

cout<<"'setprecision':"<<endl;

for(int poinplace=0; poinplace<=8; poinplace++)

cout<<setprecision(poinplace)<<theroot<<endl;

system("pause");

}
```

Field Width

- Sets the field width and returns the previous width. If values processed are smaller than the field width, fill characters are inserted as padding. Wider values will not be truncated.

- Use width() or setw(). For example:
cout.width(6); //field is 6 position wide

Program example:

```
//using width member function
```

```
#include <iostream.h>

#include <stdlib.h>

void main(void)

{

int p = 6;

char string[20];


cout<<"Using field width with setw() or
width()"<<endl; cout<<"-------------------------------------
--"<<endl; cout<<"Enter a line of text:"<<endl;


cin.width(7);

while (cin>>string)


{

cout.width(p++);

cout<<string<<endl;

cin.width(7);

//use ctrl-z followed by return key or ctrl-d to exit

}

system("pause");

}
```

Stream Format States

- Format state flag specify the kinds of formatting needed during the stream operations.
- Available member functions used to control the flag setting are: setf(), unsetf() and

flags().

- flags() function must specify a value representing the settings of all the flags.

- The one argument, setf() function specifies one or more ORed flags and ORs them with the existing flag setting to form a new format state.

- The setiosflags() parameterized stream manipulator performs the same functions as the setf.

- The resetiosflags() stream manipulator performs the same functions as the unsetf() member function. For parameterized stream manipulators you need iomanip.h header file.

- Format state flags are defined as an enumeration in class ios.

Format state flags

ios::skipws

Use to skip whitespace on input.

ios::adjustfield

Controlling the padding, left, right or internal.

ios::left

Use left justification.

ios::right

Use right justification.

ios::internal

Left justify the sign, right justify the magnitude.

ios::basefield

Setting the base of the numbers.

ios::dec

Use base 10, decimal.

ios::oct

Use base 8, octal.

ios::hex

Use base 16, hexadecimal.

ios::showbase

Show base indicator on output.

ios::showpoint

Shows trailing decimal point and zeroes.

ios::uppercase

Use uppercase for hexadecimal and scientific notation values.

ios::showpos

Shows the + sign before positive numbers.

ios::floatfield

To set the floating point to scientific notation or fixed format.

ios::scientific

Use scientific notation.

ios::fixed

Use fixed decimal point for floating-point numbers.

ios::unitbuf

Flush all streams after insertion.

ios::stdio

Flush stdout, stderr after insertion.

- skipws flags indicates that >> should skip whitespace on an input stream. The default behavior of >> is to skip whitespace. To change this, use the unsetf(ios::skipws). ws stream manipulator also can be used for this purpose.

Trailing Zeroes and Decimal Points

- ios::showpoint – this flag is set to force a floating point number to be output with its decimal point and trailing zeroes. For example, floating point 88.0 will print 88 without showpoint set and 88.000000 (or many more 0s specified by current precision) with

showpoint set.

```
///Using showpoint

//controlling the trailing zeroes and floating points
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main(void)
{
cout<<"Before using the ios::showpoint flag"
<<"-----------------------------------"<<endl;
cout<<"cout prints 88.88000 as: "<<88.88000
<<"prints 88.80000 as: "<<88.80000
<<"prints 88.00000 as: "<<88.00000
<<"using the ios::showpoint flag"
<<"-----------------------------------"<<endl;
cout.setf(ios::showpoint);
cout<<"cout prints 88.88000 as: "<<88.88000
```

```
<<"prints 88.80000 as: "<<88.80000

<<"prints 88.00000 as: "<<88.00000<<endl;

system("pause");

}
```

Justification

- Use for left, right or internal justification.
- ios::left – enables fields to be left-justified with padding characters to the right.
- ios::right – enables fields to be right-justified with padding characters to the left.
- The character to be used for padding is specified by the fill or setfill.

- internal – this flag indicates that a number‘s sign (or base if ios::showbase flag is set) should be left-justified within a field, the number‘s magnitude should be right-justified and the intervening spaces should be padded with the fill character.

- The left, right and internal flags are contained in static data member ios::adjustfield, so ios::adjustfield argument must be provided as the second argument to setf when setting the right, left or internal justification flags because left, right and internal are mutually exclusive.

```
//using setw(), setiosflags(), resetiosflags() manipulators
//and setf and unsetf member functions #include
<iostream.h>
```

```
#include <iomanip.h>
#include <stdlib.h>
void main(void)
{
long p = 123456789L;
//L - literal data type qualifier for long...
//F - float, UL unsigned integer...
cout<<"The default for 10 fields is right justified:"
<<setw(10)<<p
<<"member function"
<<"---------------------"
<<"setf() to set ios::left:"<<setw(10);
cout.setf(ios::left,ios::adjustfield);

cout<<p<<"unsetf() to restore the default:";
cout.unsetf(ios::left);
cout<<setw(10)<<p
<<"parameterized stream manipulators"
<<"---------------------------------------"
<<"setiosflags() to set the ios::left:"
<<setw(10)<<setiosflags(ios::left)<<p
<<"resetiosflags() to restore the default:"
<<setw(10)<<resetiosflags(ios::left)
<<p<<endl;
system("pause");
}
```

Padding

- fill() – this member function specify the fill character to be used with adjusted field. If no value is specified, spaces are used for padding. This function returns the prior padding

character.

- setfill() – this manipulator also sets the padding character.

```
//using fill() member function and setfill() manipulator
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
long p = 30000;
cout<<p
<<" printed using the default pad character"
<<"for right and left justified and as hex"
<<"with internal justification."
<<"-----------------------------------------";
cout.setf(ios::showbase);
cout<<setw(10)<<p<<endl;
cout.setf(ios::left,ios::adjustfield);
```

```
cout<<setw(10)<<p<<endl;

cout.setf(ios::internal,ios::adjustfield);

cout<<setw(10)<<hex<<p<<"";


cout<<"Using various padding
character"<<endl; cout<<"-------------------------
----"<<endl; cout.setf(ios::right,ios::adjustfield);


cout.fill('#');

cout<<setw(10)<<dec<<p<<";

cout.setf(ios::left,ios::adjustfield);

cout<<setw(10)<<setfill('$')<<p<<";

cout.setf(ios::internal,ios::adjustfield);

cout<<setw(10)<<setfill('*')<<hex<<p<<endl;

system("pause");


}
```

Scientific Notation

- ios::scientific and ios::fixed flags are contained in the static member ios::floatfield (usage similar to ios::adjustfield and ios::basefield).

- These flags used to control the output format of floating point numbers.

- The scientific flag – is set to force the output of a floating point number to display a specific number of digits to the right of the decimal point (specified by the precision member

function).

- cout.setf(0, ios::floatfield) restores the system default format for the floating number output.

Program example:

```
//Displaying floating number in system
//default, scientific and fixed format
#include <iostream.h>
```

```
#include <stdlib.h>
void main(void)

{

double p = 0.000654321, q = 9.8765e3;
cout<<"Declared variables" <<"--------
----------"

<<"0.000654321"<<"<<"9.8765e3"<<""; cout<<"Default format:"
<<"---------------"

<<p<<"<<q<<"<<endl;

cout.setf(ios::scientific,ios::floatfield);
cout<<"Scientific format:" <<"--------
----------"

<<p<<"<<q<<";

cout.unsetf(ios::scientific);
cout<<"format after unsetf:" <<"---
------------------------"
<<p<<"<<q<<endl;
cout.setf(ios::fixed,ios::floatfield);
cout<<"fixed format:" <<"----------
------" <<p<<"<<q<<endl;
```

```
system("pause");
}
```

File Handling

Opening a File: *fopen*

- We're able to create, modify, and access files easily from PHP.

Files are used to store customer data, store page hit counts, remember end-user preferences, and a plethora of other things in web applications.

- The fopen function opens a file for reading or writing.

fopen (string *filename*, string *mode*,

[, int *use_include_path*

[, resource *zcontext*]])

filename - name of the file including path

mode - how the file will be used read, write, etc

use_include_path - optional parameter that

can be set to 1 to specify

that you want to search for

the file in the PHP include

path

zcontext - not covered in these notes

File Modes

•'r' - open for reading only

•'r+' - open for reading & writing

•'w' - open for writing only, create file if

necessary

•'w+' - open for reading and writing, create file

if necessary

•'a' - open for appending, (start writing at

end of file)

•'a+' - open for reading and writing

•'x' - create and open for writing only, if file

already exists fopen will fail

•'x+' - create and open for reading and

writing

•Note -------

different operating systems have different

line-ending conventions. When inserting a line-break into a text file you need to use the correct line-ending character(s). Windows systems use , unix based systems use .

PHP has a constant PHP_EOL which holds the correct one for the system you are using.

File Open Examples

•In Windows, you can use a text-mode translation flag ('t'), which will translate to when

working with the file. To use this flag specify 't' as the last character of the mode parameter, i.e.; 'wt'.

•Examples

-- open the file /home/file.txt for reading

$handle = fopen("/home/file.txt", "r");

after this you'll refer to the file with the variable

$handle -- open a file for writing

$handle = fopen("/home/wrfile.txt","w");

the file will be created if it doesn't exist,

if it does exist you will overwrite whatever is currently in the
file File Open Examples

•open a file for binary writing

$handle = fopen("/home/file.txt", "wb");

•In Windows you should be careful to escape any backslashes used in the path to the file

$handle = fopen("c:\data\file.txt", "r");

•you can open files on another system

$handle = fopen("http://www.superduper.

com/file.txt", "r");

•can use ftp

```
$handle = fopen("ftp://user:password@
superduper.com/file.txt",
"r");
```

Reading Lines of Text: *fgets*

•The fgets function reads a string of text from a file.

fgets (resource handle [, int length]);

Pass this function the file handle to an open file, and an optional length. Reading ends when length - 1 bytes have been read, on a newline (which is included in the return value), or on encountering the end of file, whichever comes first. If no length is specified, the default length is 1024 bytes. (file.txt below)

```
Here
is
the
file.
```

```
$handle = fopen("file.txt","r");
```

```
while (!feof($handle))
{
$text = fgets ($handle);
}
fclose($handle);
```

Reading Characters: fgetc

•The fgetc function let's you read a single character from an open file.

Here's
the file
contents.

```
$fhandle = fopen("file.txt", "r");

while ($char = fgetc($fhandle))

{
if ($char == "")
{
$char = "<br />";
}
echo "$char";
}

fclose($fhandle);
```

•Binary Reading: fread

•Files don't have to be read line by line, you can read a specified number of bytes (or until the end-of-file is reached). The file is treated as a simple binary file of bytes.

fread (resource handle, int length);

Bytes are read up to the length specified or EOF is reached. On Windows systems, you should open files for binary reading (mode 'rb') to work with fread. Adding 'b' to the mode does no harm on other systems, it can be included for portability.

```
$handle = fopen("file.txt","rb");

$text = fread($handle, filesize("file.txt");


//the filesize function returns the no. of bytes
//in the file.
```

you can convert line endings to

```
$br_text = str_replace("", "<br />", $text);
```

•Reading a Whole File: file_get_contents

•The file_get_contents function will read the entire contents of a file into a string. No file handle is needed for this function.

```
$text = file_get_contents("file.txt");
```

$br_text = str_replace("","
",$text);

echo $br_text;

Careful with this function, for very large files, it can be a problem since the entire file must be memory resident at one time.

Parsing a File

To make it easier to extract data from a file, you can format that file (using, for example, tabs) and use fscanf to read your data from the file.

In general -----

fscanf (*resource handle*, *string format*);

•Parsing a file: fscanf (cont.)

•This function takes a file handle, handle, and format, which describes the format of the file you're working with. You set up the format in the same way as with sprintf, which was

discussed in Chapter 3. For example, say the following data was contained in a file names

tabs.txt, where the first and last names are separated by a tab.

George Washington

Benjamin Franklin

Thomas Jefferson

$fh = fopen("tabs.txt", "rb");

while ($names = fscanf($fh, "%s%s"))

{

```
list($firstname,$lastname) = $names;

echo $firstname, " ", $lastname, "<br />";

}

fclose ($fh);
```

Writing to a File: fwrite

•Write to a file with fwrite.

fwrite (resource *fh*, string *str* [, int *length*]);

The fwrite function writes the contents of *str* to the file stream represented by *fh*. The writing will stop after *length* bytes have been written or the end of the string is reached.

fwrite returns the number of bytes written or FALSE if there was an error. If you're working on a Windows system the file must be opened with 'b' included in the fopen mode parameter.

```
$fh = fopen("text.txt","wb");

$text = "Here.";

if (fwrite($fh, $text) == FALSE)

{

echo "Cannot write to text.txt.";

}

else {

echo "Created the file text.txt.";
```

}

fclose($fh);

Appending to a File: fwrite

•In this case open the file for appending, using the file mode 'a':

$fh = fopen("text.txt","ab");

Append this to text.txt:

And here is

more text.

$text = "here istext.";

if (fwrite($fh, $text) == FALSE)

{

echo "Cannot write to text.txt.";

}

else {

echo "Appended to the file text.txt.";

}

fclose($fh);

Writing a File at Once: file_put_contents

•This function writes a string to a file, and here's how you use it in general:

file_put_contents (string *filename*, string *data*

[, int *flags*

[,resource *context*]]);

```
$text = "Here.";

if (file_put_contents("text.txt",$text) ==
FALSE)
{
echo "Cannot write to text.txt.";
}
else {
echo "Wrote to the file text.txt";
}
```

Random Access

This lesson is about using random access files in C and the following lesson will look at working with text files. Apart from the simplest of applications, most programs have to read or write files. Maybe it's just for reading a config file, or a text parser or something more sophisticated. The basic file operations are

fopen - open a file- specify how its opened (read/write) and type (binary/text)

fclose - close an opened file

fread - read from a file

fwrite - write to a file

fseek/fsetpos - move a file pointer to somewhere in a file.

ftell/fgetpos - tell you where the file pointer is located.

There are two fundamental types of file: text and binary. Of these two, binary are generally the simpler to deal with. As doing random access on a text file isn't something you need to do too often, we'll stick with binary files for the rest of this lesson. The first four operations listed above are for both text and random access files. The last two just for random access.

Random access means we can move to any part of a file and read or write data from it without having to read through the entire file. Back thirty years ago, much data was stored on large reels of computer tape. The only way to get to a point on the tape was by reading all the way through the tape. Then disks came along and now we can read any part of a file directly.

Object Serialization

The C++ language provides a somewhat limited support for file processing. This is probably based on the time it was conceived and put to use. Many languages that were developed after C++, such as (Object) Pascal and Java provide a better support, probably because their libraries were implemented as the demand was made obvious. Based on this, C++ supports saving only values of primitive types such as short, int, char double. This can be done by using either the C FILE structure or C++' own fstream class.

Binary Serialization

Object serialization consists of saving the values that are part of an object, mostly the value gotten from declaring a variable of a class. AT the current standard, C++ doesn't inherently support object serialization. To perform this type of operation, you can use a technique

known as binary serialization.

When you decide to save a value to a medium, the **fstream** class provides the option to save the value in binary format. This consists of saving each byte to the medium by aligning bytes in a contiguous manner, the same way the variables are stored in binary numbers.

To indicate that you want to save a value as binary, when declaring the **ofstream** variable, specify the **ios** option as **binary**. Here is an example:

```
#include <fstream>

#include <iostream>

using namespace std;

class Student

{

public:


char FullName[40];


char CompleteAddress[120];


char Gender;
```

```
double Age;

bool LivesInASingleParentHome;

};

int main()
{
	Student one;
	strcpy(one.FullName, "Ernestine Waller");
	strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");
	one.Gender = 'F';

	one.Age = 16.50;
	one.LivesInASingleParentHome = true;
	 ofstream ofs("fifthgrade.ros", ios::binary);
	return 0;
}
```

Writing to the Stream

The **ios::binary** option lets the compiler know how the value will be stored. This declaration also initiates the file. To write the values to a stream, you can call the **fstream::write()** method.

After calling the write() method, you can write the value of the variable to the medium. Here is an example:

```
#include <fstream>
#include <iostream>
```

```
using namespace std;
class Student
{
public:
	char FullName[40];
	char CompleteAddress[120];
	char Gender;
	double Age;
	bool LivesInASingleParentHome;
};
int main()
{

Student one;
	strcpy(one.FullName, "Ernestine Waller");
	strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");
	one.Gender = 'F';
	one.Age = 16.50;
	one.LivesInASingleParentHome = true;
	ofstream ofs("fifthgrade.ros", ios::binary);
	ofs.write((char *)&one, sizeof(one));
	return 0;
```

```
}
```

Reading From the Stream

Reading an object saved in binary format is as easy as writing it. To read the value, call the ifstream::read() method. Here is an example:

```
#include <fstream>
#include <iostream>
using namespace std;
class Student
{
public:
	char FullName[40];
	char CompleteAddress[120];

	char Gender;
	double Age;
	bool LivesInASingleParentHome;
};
int main()
{
/*
Student one;
	strcpy(one.FullName, "Ernestine Waller");
	strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");
	one.Gender = 'F';
	one.Age = 16.50;
	one.LivesInASingleParentHome = true;
```

```
ofstream ofs("fifthgrade.ros", ios::binary);

        ofs.write((char *)&one, sizeof(one));


*/

        Student two;

        ifstream ifs("fifthgrade.ros", ios::binary);

        ifs.read((char *)&two, sizeof(two));

        cout << "Student Information";


        cout << "Student Name: " << two.FullName << endl;
        cout << "Address: " << two.CompleteAddress << endl;
        if( two.Gender == 'f' || two.Gender == 'F' ) cout <<
        "Gender: Female" << endl;


        else if( two.Gender == 'm' || two.Gender == 'M' )

        cout << "Gender: Male" << endl;

        else

        cout << "Gender: Unknown" << endl;

        cout << "Age: " << two.Age << endl;

        if( two.LivesInASingleParentHome == true )

                cout << "Lives in a single parent home" << endl;

        else
```

```
cout << "Doesn't live in a single parent home" << endl;

cout << "";

return 0;


}
```

NAMESPACE:

- Namespace: a naming context
- Conflicts are bad
- Make names unique within a namespace, make namespaces unique
- AC ++ namespace: contains classes, variables, constants, functions, etc.
- Names in a namespace are visible outside the namespace
- Example: a class is a namespace

```
class Example {

public:

void PublicFunction ();

private:

void PrivateFunction ();

};
```

- Both PublicFunction() and PrivateFunction() are visible outside the class, as Example::PublicFunction() and Example::PrivateFunction()

- Need to distinguish between different items with the same name
- Example: two libraries both have a String class
- C solution: make the names different (Library1String and Library2String)
- Requires vendors to cooperate, and makes all names longer (kThemeWidgetCloseBox)
- Bad C++ solution: use dummy classes or structs to group names

```
struct Library1 {

static void Function1 ();

};

struct Library2 {
```

```
static void Function1 (int);

};
```

- Requires the entire library to be in one header file (or included from it)
- Everything not in a class or in a function has to be global

Namespace syntax

```
namespace MyNamespace {

void Function1 ();

void Function2 ();

typedef UInt32 MyInt32;

MyInt32 gVariable;

}
```

- Very similar to class syntax
- No access specification (public, private, protected)
- No trailing semicolon

- Namespaces are open – they can be in several independent header files Library1String.h:

```
namespace Library1 {

class String;

}
```

Library1List.h:

```
namespace Library1 {

class List;

}
```

Using a namespace

```
namespace Library1 {

class String;

class List;

}
```

- Explicit qualification

```
void DoSomething ()

{


Library1::String string;
Library1::List list;


// ...

}
```

- using declaration

```
void DoSomething ()
{


using Library1::String;
using Library1::List;
String string;


List list;
// ...
```

}

- using directive

void DoSomething ()

{

using namespace Library1;

String string;

List list;

}

Nested namespaces

namespace Library1 {

namespace Part1 {

class String;

class List;

}

namespace Part2 {

class String;

class List;

class Array;

}

}

Ansi string objects

§

The ANSI string class implements a first-class character string data type that avoids many problems

§

associated with simple character arrays ("C-style strings"). You can define a string object very

§

simply, as shown in the following example

```
#include <string>
using namespace std;
...
string first_name = "Bjarne";
string last_name;
last_name = "Stroustrup";
string names = first_name + " " + last_name;
cout << names << endl;
names = last_name + ", " + first
" + first_name;
cout << names << endl;
```

Member functions

§

The string class defines many member functions. A few of the basic ones are

described below:

§

A string object may defined without an initializing value, in which case its initial

§

value is an empty string (zero length, no characters):

§

string str1;

§

A string object may also be initialized with

§


a string expression:
§

string str2 = str1;

§

string str3 = str1 + str2;

§

string str4 (str2); // Alternate form

§


a character string literal:
§

string str4 = "Hello there";

Standard Template Library

§

The standard template library (STL) contains

§

Containers

§

Algorithms

§

Iterators

§

A *container* is a way that stored data is organized in memory, for example an array of elements.

§

Algorithms in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.

§

Iterators are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array

Part-A (2-Marks)

1. Define Stream.

2. What is meant by namespaces?
3. Define setf().
4. Give the use of putback() and peek() function.
5. Define Fileobject.
6. What is meant by STL?
7. Define Object Serialization.
8. Name some File Modes?
9. Define Manipulators.
10. What are the types of Manipulators?
11. Give the use of ios::showbase.
12. What is meant by ifstream?
13. Write the syntax for File open and close.
14. What is the need of STL?
15. List out the advantages and disadvantages of STL.
16. List out the STL Containers.

Part-B

1.Explain in detail about Formatted IO in C++. (16)

2.Explain about STL in detail with necessary example. (16)

3. a. Write a program in C++ to perform binary serialization using file streams.
(8) b. Write a program in C++ to demonstrate the use of std namespaces. (8)

4.Explain in detail about Manipulators. (16)

5. Explain the hierarchy of File Stream Classes. (16)

6. a. Write a program to read and write a File.
(8) Object Oriented Programming 5

Kings College Of Engineering, Punalkulam.

b.Write a program to use set precision and precision manipulators. (8)

Packages and Interfaces, Exception handling, Multithreaded programming, Strings, Input/Output

1.Function and Class Templates:

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates.

One good area where this C++ Class Templates are suited can be container classes. Very famous examples for these container classes will be the STL classes like vector, list etc.,

Once code is written as a C++ class template, it can support all data types. Though very

useful, It is advisable to write a class as a template after getting a good hands-on experience on the logic .

Declaring C++ Class Templates:

Declaration of C++ class template should start with the keyword *template*. A parameter should be included inside angular brackets. The parameter inside the angular brackets, can be either the keyword *class* or *typename*.

This is followed by the class body declaration with the member data and member functions. The following is the declaration for a sample Queue class.

```
//Sample code snippet for C++ Class Template

template < typename T>
```

```
class MyQueue

{

std::vector<T> data;

public:

void Add(T const &d);

void Remove();

void Print();

};
```

The keyword class highlighted in blue color, is not related to the *typename*. This is a mandatory keyword to be included for declaring a template class.

Defining member functions - C++ Class Templates:

If the functions are defined outside the template class body, they should always be defined with the full template definition. Other conventions of writing the function in C++ class templates are the same as writing normal c++ functions.

```
template < typename T> void MyQueue<T> ::Add(T const &d)

{

data.push_back(d);

}


template < typename T> void MyQueue<T>::Remove()

{

data.erase(data.begin( ) + 0,data.begin( ) + 1);

}


template < typename T> void MyQueue<T>::Print()

{
```

```
std::vector <int>::iterator It1;

It1 = data.begin();

for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )

cout << " " << *It1<<endl;


}
```

The Add function adds the data to the end of the vector. The remove function removes the first element. These functionalities make this C++ class Template behave like a normal Queue. The print function prints all the data using the iterator.

Full Program - C++ Class Templates:

```
//C++_Class_Templates.cpp


#include <iostream.h>

#include <vector>


template <typename T>

class MyQueue

{

std::vector<T> data;
```

```
public:
void Add(T const &);
void Remove();
void Print();
};

template <typename T> void MyQueue<T> ::Add(T const &d)
{
data.push_back(d);
}

template <typename T> void MyQueue<T>::Remove()
{

data.erase(data.begin( ) + 0,data.begin( ) + 1);
}

template <typename T> void MyQueue<T>::Print()
{
std::vector <int>::iterator It1;
It1 = data.begin();
for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )
cout << " " << *It1<<endl;

}
//Usage for C++ class templates
void main()
```

```
{
MyQueue<int> q;
q.Add(1);
q.Add(2);

cout<<"Before removing data"<<endl;
q.Print();

q.Remove();
cout<<"After removing data"<<endl;
q.Print();

}
```

Advantages of C++ Class Templates:

One C++ Class Template can handle different types of parameters.

Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.

Templates reduce the effort on coding for different data types to a single set of code.

Testing and debugging efforts are reduced.

Exeption Handling:

Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an exception object. In order to deal with the exceptional situation you throw the exception. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a handler. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is caught. A handler may rethrow an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

try blocks

catch blocks

throw expressions

Exception specifications (C++ only)

try blocks (C++ only)

You use a try block to indicate which areas in your program that might throw exceptions you want to handle immediately. You use a function try block to indicate that you want to detect exceptions in the entire body of a function.

```
#include <iostream>

using namespace std;
```

```
class E {

public:

const char* error;

E(const char* arg) : error(arg) { }

};


class A {


public:

int i;


//A function try block with a member
//initializer
A() try : i(0) {

throw E("Exception thrown in A()");

}

catch (E& e) {

cout << e.error << endl;

}
```

```
};


//A function try
block void f() try {



throw E("Exception thrown in f()");

}

catch (E& e) {

cout << e.error << endl;

}


void g() {

throw E("Exception thrown in g()");

}


int main() {

f();


//A try
block try {

g();

}

catch (E& e) {

cout << e.error << endl;

}


try {
A x;
```

```
}

catch(...) { }

}
```

The following is the output of the above example:
Exception thrown in f()

Exception thrown in g()
Exception thrown in A()

The constructor of class A has a function try block with a member initializer. Function f() has a function try block. The main() function contains a try block.

catch blocks (C++ only)

catch block syntax

```
>>-catch--(--exception_declaration--)--{--statements--}-------><
```

You can declare a handler to catch many types of exceptions. The allowable objects that a function can catch are declared in the parentheses following the catch keyword (the

exception_declaration). You can catch objects of the fundamental types, base and derived class objects, references, and pointers to all of these types. You can also catch const and volatile types. The *exception_declaration* cannot be an incomplete type, or a reference or pointer to an incomplete type other than one of the following:

void*

const void*

volatile void*

const volatile void*

You cannot define a type in an *exception_declaration*.

You can also use the catch(...) form of the handler to catch all thrown exceptions that have not been caught by a previous catch block. The ellipsis in the catch argument indicates that any exception thrown can be handled by this handler.

If an exception is caught by a catch(...) block, there is no direct way to access the object thrown. Information about an exception caught by catch(...) is very limited.

You can declare an optional variable name if you want to access the thrown object in the catch block.

A catch block can only catch accessible objects. The object caught must have an accessible copy constructor.

throw expressions (C++ only)

You use a throw expression to indicate that your program has encountered an exception.
throw expression syntax

>>-throw--+----------------------+---------------------------><

'-assignment_expression-'

The type of *assignment_expression* cannot be an incomplete type, or a pointer to an incomplete type other than one of the following:

void*

const void*

volatile void*

const volatile void*

The *assignment_expression* is treated the same way as a function argument in a call or the operand of a return statement.

If the *assignment_expression* is a class object, the copy constructor and destructor of that object must be accessible. For example, you cannot throw a class object that has its copy constructor declared as private.

Exception specifications (C++ only)

C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function

acts as a guarantee to the function's caller that the function will throw only the exceptions contained in the exception specification.

For example, a function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

explicitly states that it will only throw exception objects whose types are unknown_word or bad_grammar, or any type derived from unknown_word or bad_grammar.

Exception specification syntax

```
>>-throw--(--+--------------+--)------------------------------><
            '-type_id_list-'
```

The *type_id_list* is a comma-separated list of types. In this list you cannot specify an incomplete type, a pointer or a reference to an incomplete type, other than a pointer to void, optionally qualified with const and/or volatile. You cannot define a type in an exception specification.

A function with no exception specification allows all exceptions. A function with an exception specification that has an empty *type_id_list*, throw(), does not allow any

exceptions to be thrown.

Function f() can throw objects of types A or B. If the function tries to throw an object of type C, the compiler will call unexpected() because type C has not been specified in the function's exception specification, nor does it derive publicly from A. Similarly, function g() cannot throw pointers to objects of type C; the function may throw pointers of type A or pointers of objects that derive publicly from A.

A function that overrides a virtual function can only throw exceptions specified by the virtual function. The following example demonstrates this:

```
class A {
public:
virtual void f() throw (int, char);
};

class B : public A{
public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
```

```
class C : public A {
public: void f() { }
};

class D : public A {
public: void f() throw (int, char, double) { }
};
*/
```

The compiler allows B::f() because the member function may throw only exceptions of type

int. The compiler would not allow C::f() because the member function may throw any kind of exception. The compiler would not allow D::f() because the member function can throw more types of exceptions (int, char, and double) than A::f().

Implicitly declared special member functions (default constructors, copy constructors, destructors, and copy assignment operators) have exception specifications. An implicitly declared special member function will have in its exception specification the types

declared in the functions' exception specifications that the special function invokes. If any function that a special function invokes allows all exceptions, then that special function allows all exceptions. If all the functions that a special function invokes allow no exceptions, then that

special function will allow no exceptions. The following example demonstrates this:

```
class A {
public:
A() throw (int);
A(const A&) throw (float);
~A() throw();
};

class B {
public:
B() throw (char);
B(const A&);
~B() throw();

};

class C : public B, public A { };
```

The following special functions in the above example have been implicitly declared:

```
C::C() throw (int, char);

C::C(const C&); // Can throw any type of exception, including
float C::~C() throw();
```

terminate() and unexpected()

Earlier in this issue the basic purposes of the terminate() and unexpected() functions are described. In the past year the standards committee has made several refinements to these functions.

The committee has confirmed that direct calls may be made to these functions from

application code. So for instance:

```
#include <exception>

        if (something_is_really_wrong)

std::terminate();
```

This will terminate the program without unwinding the stack and destroying local (and finally static) objects. Alternatively, if you just throw an exception that doesn't get handled, it is implementation- dependent whether the stack is unwound before terminate() is called. (Most implementations will likely support a mode wherein the stack is not unwound, so that you can debug from the real point of failure).

Probably the main purpose of making direct calls to terminate() and unexpected() will be to simulate possible error conditions in application testing, especially when the application has established its own terminate and unexpected handlers.

The committee has changed slightly the definition of what handlers are used when terminate() or unexpected() are called. In most cases, they are now the handlers in effect at

the time of the throw, which are not necessarily the current handlers. Usually they are one and the same, but consider:

```
#include <exception>


void u1() { ... }
```

```
void u2() { ... }


class A {

public:

A() { ... }

A(const A&) { ... std::set_unexpected(u2); }

};


void f() throw(int)

{

A a;

throw a; // which unexpected handler gets called?

}


int main()

{

std::set_unexpected(u1);

f();

return 0;

}
```

The copy constructor for A is called as part of the throw operation in f(), so by the time the C++ implementation determines that an unexpected handler needs to be called, u2() is the

current handler. However, based on this recent change, it is the handler in effect at the time of the throw - u1() - which gets called. On the other hand, if a direct call to terminate() or unexpected() is made from the application, it is always the current handler which gets called.

Some would argue that this kind of rule just adds complexity without much benefit to already-complex C++ implementations, but others feel that if an application is going to be

dynamically changing its terminate and unexpected handlers, retaining the correct association is important.

In the next issue we'll talk about another clarification of terminate() and unexpected(), this time related to the uncaught_exception() library function introduced above.

Uncaught exceptions:

In the past few examples, there are quite a few cases where a function assumes its caller (or another function somewhere up the call stack) will handle the exception. In the following example, MySqrt() assumes someone will handle the exception that it throws — but what

happens if nobody actually does?

Here‘s our square root program again, minus the try block in main():

```
#include "math.h" // for sqrt() function

using namespace std;

A modular square root function

double MySqrt(double dX)

{

//If the user entered a negative number, this is an error
condition if (dX < 0.0)

}

return sqrt(dX);

throw "Can not take sqrt of negative number"; // throw exception of type char*
```

```
int main()

{

}

double dX;

cin >> dX;

cout << "The sqrt of " << dX << " is " << MySqrt(dX) << endl; cout << "Enter a number: ";
```

MySqrt() doesn‘t handle the exception, so the program stack unwinds and control returns to main(). But there‘s no exception handler here either, so main() terminates. At this point, we just terminated our application!

When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred. How it does this depends on the operating system, but possibilities include printing an error message, popping up an error dialog, or simply crashing. Some OS‘s are less graceful than others. Generally this is something you want to avoid altogether!

Catch-all handlers

And now we find ourselves in a condundrum: functions can potentially throw exceptions of any data type, and if an exception is not caught, it will propagate to the top of your program and cause it to terminate. Since it‘s possible to call functions without knowing how they are even implemented, how can we possibly prevent this from happening?

Fortunately, C++ provides us with a mechanism to catch all types of exceptions. This is known as a **catch-all handler**. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

ONLINE QUESTIONS
UNIT-I

questions	opt1	opt2	opt3	opt4			answer
Multiple inheritance means,	one class inheriting from more super classes	A private member of a class cannot be accessed by the methods of the same class	None of the above	(a) and (b) above.			A private member of a class cannot be accessed by the methods of the same class
Which statement is not true in java language?	static	const	final	none of the above			final
Which one of the following is not true?	A class containing abstract methods is called an abstract class	Abstract methods should be implemented in the derived class	An abstract class cannot have non-abstract methods	A class must be qualified as 'abstract' class, if it contains one abstract method			An abstract class cannot have non-abstract methods
What is byte code in the context of Java	The type of code generated by a Java compiler	The type of code generated by a Java Virtual Machine	It is another name for a Java source file	It is the code written within the instance methods of a class			The type of code generated by a Java compiler
The correct order of the declarations in a Java program is	Package declaration, import statement, class declaration	Import statement, package declaration, class declaration	Import statement, class declaration, package declaration	Class declaration, import statement, package declaration			Package declaration, import statement, class declaration
Mark the incorrect statement from the following:	Java is a fully object oriented language with strong support for proper software engineering techniques	In java it is not easy to write C-like so called procedural programs	In java language objects have to be manipulated	In java language error processing is built into the language			In java language error processing is built into the language
Which of the following is not a component of Java Integrated Development Environment (IDE)?	Net Beans	Borland's Jbuilder	Microsoft Visual Fox Pro	Symantec's Visual Café			Microsoft Visual Fox Pro
Java compiler javac translates Java source code into ……	Assembler language	Byte code	Bit code	Machine code			Byte code
In object-oriented programming, the process by which one object acquires the properties of another object is called	Encapsulation	Polymorphism	Overloading	Inheritance			Inheritance

A process that involves recognizing and focusing on the important characteristics of a situation or object is known as:	Encapsulation	Polymorphism	Abstraction	Inheritance			Abstraction
Object oriented inheritance models the	"is a kind of" relationship	"has a" relationship	"want to be" relationship	inheritance does not describe any kind of relationship between classes			"is a kind of" relationship
The wrapping up of data and functions into a single unit is called	Encapsulation	Abstraction	Data Hiding	Polymorphism			Encapsulation
In object oriented programming new classes can be defined by extending existing classes. This is an example of:	Encapsulation	Inheritance	Aggregation	Interface			Inheritance
Basic Java language functions are stored in which of the following java package?	java.lang	java.io	java.net	java.util			java.lang
What is the fundamental unit of information of writer streams?	Characters	Bytes	Files	Records			Characters
URL stands fo	Universal reader locator	Universal reform loader	Uniform resource loader	Uniform resource locator			Characters
Which of the following concept is not there in Java?	Encapsulation	Operator Overloading	Data Hiding	Data Hiding			Operator Overloading
Which of the following concept is not there in Java?	Encapsulation	Data Hiding	Multiple inheritance	Abstraction			Multiple inheritance
Which Browser supports Java?	Chrome	Internet Explorer	Hotjava	All the above			All the above
Which Operation system supports Java?	Sun Solaris	RedHat Linux 3.0	Windows Server 2003	All the above			All the above
Which is not part of Java development Kit?	jbc	javah	encoder	javac			encoder
Whis is not part of Java Runtime Environment?	Java Virtual Machine	Runtime class livraries	User interface toolkits	decrypter			decrypter
What is dynamic Binding?	Object creation at run time	Linking of Procedure call at run time	Class creation at run time	none of the above			Linking of Procedure call at run time
Which is not a benefit of OOP?	Reduction in complexity	Possible to have multiple objects	Data centered design approach	Not possible to create windows application			Not possible to create windows application
Java was developed by?	Oracle	Microsoft	Sun	Ericsson			Sun

			Microsystems				Microsystems
Initial name given to Java was?	apple	Oak	foxpro	access			Oak
Java doesn't support?	overloading	Sizeof	inheritance	Interface			Sizeof
Which preprocessor directive cann't be used in Java?	#define	#include	#ifdef	all the above			all the above
First application written in Java was?	MSC OS	Unix	HotJava	Sun Browser			HotJava

UNIT-II

Questions	Choice 1	Choice 2	Choice 3	Choice 4			Answer
Which of these keywords is used to make a class?	class	struct	int	None of the mentioned			None of the mentioned
Which of these operators is used to allocate memory for an object?	malloc	alloc	new	give			New
What is the return type of a method that does not returns any value?	int	float	void	double			void
What is the process of defining more than one method in a class differentiated by method signature?	Function overriding	Function overloading	Function doubling	None of the mentioned			Function overloading
Which of the following is a method having same name as that of it's class?	finalize	delete	class	constructor			constructor
Which method can be defined only once in a program?	main method	finalize method	static method	private method			main method
Which of these access specifiers must be used for main() method?	private	public	protected	None of the mentioned			public
Which of these is used to access member of class before object of that class is created?	public	private	static	protected			static
What is the process by which we can control	Polymorphism	Abstraction	Encapsulation	Recursion			Encapsulation

what parts of a program can access the members of a class?							
What is process of defining two or more methods within same class that have same name but different parameters declaration?	method overloading	method overriding	method hiding	None of the mentioned			method overloading
Which of these can be overloaded?	Methods	Constructors	All of the mentioned	None of the mentioned			All of the mentioned
What is the process of defining a method in terms of itself, that is a method that calls itself?	Polymorphism	Abstraction	Encapsulation	Recursion			Recursion
Which of these keywords is used to refer to member of base class from a sub class?	upper	super	this	None of the mentioned			super
A class member declared protected becomes member of subclass of which type?	public member	private member	protected member	static member			private member
Which of these is correct way of inheriting class A by class B?	class B + class A {}	class B inherits class A {}	class B extends A {}	class B extends class A {}			class B extends A {}
Which of these keyword can be used in subclass to call the constructor of superclass?	super	this	extent	extends			super
What is the process of defining a method in subclass having same name & type signature as a method in its superclass?	Method overloading	Method overriding	Method hiding	None of the mentioned			Method overriding
Which of these keywords can be used to prevent Method overriding?	static	constant	protected	final			final
Which of these is correct way of calling a constructor having no parameters, of superclass A by subclass B?	super(void);	superclass.();	super.A();	super();			super();

Which of these is supported by method overriding in Java?	Abstraction	Encapsulation	Polymorphism	None of the mentioned			Polymorphism
Which of these class relies upon its subclasses for complete implementation of its methods?	Object class	abstract class	ArrayList class	None of the mentioned			abstract class
Which of these operators can skip evaluating right hand operand?	!	|	&	&&			&&
Which of these have highest precedence?	()	++	*	>>			()
Which of these is returned by greater than, <, and equal to, ==, operator?	Integers	Floating - point numbers	Boolean	None of the mentioned			Boolean

class Relational_operator {
public static void main(String args[])
{
int var1 = 5;
int var2 = 6;
System.out.print(var1 > var2);
}
}
	1	0	TRUE	FALSE			FALSE

class Output {
public static void main(String args[])
{
int x , y = 1;
x = 10;
if (x != 10 && x / 0 == 0)
 	
System.out.println(y);
else
 	
System.out.println(++y);
}
}
	1	2	Runtime error owing to division by zero in if condition	Unpredictable behavior of program			2

int ++a = 100 ; System.out.println(++a) ; What will be the output of the above fraction of code ?	100	101	Compiler displays error as ++a is not a valid identifier	None of these			Compiler displays error as ++a is not a valid identifier

What will be the output?
if(1 + 1 + 1 + 1 + 1 == 5){
 	
System.out.print("TRUE");
}
else{
 	
System.out.print("FLASE");
}
	TRUE	FALSE	Compiler Error	None Of these			TRUE

Which one of the following is the Suggested Section in Java programming Structure	package statement	Documentation Section	Import Statement	Interface Statement			Documentation Section
Which one of the following is the Essential Section in Java programming Structure	package statement	Documentation Section	Import Statement	main method Class			main method Class
The first Statement allowed in java file is a _____ Statement.	package statement	Documentation Section	Import Statement	main method Class			package statement
Which is not a type of java token.	Identifiers	Literals	Operatore	Terminators			Terminators
Whether Identifiers in java can be of any length	TRUE	FALSE	Both	No Answer			TRUE

UNIT-III

Questions	Choice 1	Choice 2	Choice 3	Choice 4			Answer
How many primitive data types are there in Java?	6	7	8	9			8
In Java byte, short, int and long all of these are Size of int in Java is	signed	unsignes	all of the above	none of the above			signed
The smallest integer type is ..... and its size is ..... bits.	short, 8	short, 16	byte, 8	byte, 16			byte, 8
Which of the following automatic type conversion will be possible?	short to int	byte to int	int to long	long to int			int to long
double STATIC = 2.5 ; System.out.println(STATIC);	Prints 2.5	Rraises an error as STATIC is used as a variable which is a keyword	Raises an exception	None of these			Prints 2.5
In Java, the word true is	Java keyword	Boolean literal	Same as value 1	Same as value 0			Boolean literal

What is the range of data type short in Java?	-128 to 127	-32768 to 32767	-2147483648 to 2147483647	None of the mentioned			-32768 to 32767
Which of the following are legal lines of Java code?	int w = (int)888.8;	byte x = (byte)100L;	long y = (byte)100;	byte z = (byte)100L;			byte z = (byte)100L;
An expression involving byte, int, and literal numbers is promoted to which of these?	int	long	byte	float			int
Which data type value is returned by all transcendental math functions?	int	float	double	long			double
Which of the following can be operands of arithmetic operators?	Numeric	Boolean	Characters	Both Numeric & Characters			Both Numeric & Characters
Modulus operator, %, can be applied to which of these?	Integers	Floating – point numbers	Both Integers and floating – point numbers	None of the mentioned			Both Integers and floating – point numbers.
What is the output of relational operators?	Integer	Boolean	Characters	Double			Boolean
Which of these have highest precedence?	()	++	*	>>			()
What should be expression1 evaluate to in using ternary operator as in this line? expression1 ? expression2 : expression3	Integer	Floating – point numbers	Boolean	None of the mentioned			Boolean
Which of these statements are incorrect?	Equal to operator has least precedence.	Brackets () have highest precedence.	Division operator, /, has higher precedence than multiplication operator.	Addition operator, +, and subtraction operator have equal precedence.			Division operator, /, has higher precedence than multiplication operator.
Which of these is necessary condition for automatic type conversion in Java?	The destination type is smaller than source type.	The destination type is larger than source type.	The destination type can be larger or smaller than source type.	None of the mentioned			The destination type is larger than source type.
What is the prototype	prototype( )	prototype(void)	public	public			public

of the default constructor of this class? public class prototype { }			prototype(void)	prototype( )			prototype( )
What is the error in this code? byte b = 50; b = b * 50;	b can not contain value 100, limited by its range.	* operator has converted b * 50 into int, which can not be converted to byte without casting.	b can not contain value 50.	No error in this code			* operator has converted b * 50 into int, which can not be converted to byte without casting.
If an expression contains double, int, float, long, then whole expression will promoted into which of these data types?	long	int	double	float			double
What is Truncation is Java?	Floating-point value assigned to an integer type.	Integer value assigned to floating type.	Floating-point value assigned to an Floating type.	Integer value assigned to floating type.			Floating-point value assigned to an integer type.
Which of these selection statements test only for equality?	if	switch	if & switch	None of the mentioned			switch
The control expression in an "if" statement must be:	an expression with type integer	an expression with either the type boolean or integer	an expression with either the type boolean or integer with value 0 or 1	an expression with type boolean			an expression with type boolean
Which operator is used to invert all the digits in binary representation of a number?	~	>>>	^	!			~
On applying Left shift operator, <<, on an integer bits are lost one they are shifted past which position bit?	1	32	33	31			31
Which right shift operator preserves the sign of the value?	>>	>>=	<<=	none of the above			>>
What is the output of relational operators?	Integer	Boolean	Characters	Double			Boolean
Which of these is	Integers	Floating - point	Boolean	None of the			Boolean

returned by greater than, <, and equal to, ==, operator?		numbers		mentioned			
Which of these operators can skip evaluating right hand operand?	!	|	&	&&			&&
Which of these statement is correct?	true and false are numeric values 1 and 0.	true and false are numeric values 0 and 1.	true is any non zero value and false is 0.	true and false are non numeric values.			true and false are non numeric values.
Which of these statements are incorrect?	Equal to operator has least precedence.	Brackets () have highest precedence.	Division operator, /, has higher precedence than multiplication operator.	Addition operator, +, and subtraction operator have equal precedence.			Division operator, /, has higher precedence than multiplication operator.
Which of these class is superclass of all other classes?	Math	Process	System	Object			Object
When the operators are having the same priority, they are evaluated from .... in the order they appear in the expression.	right to left	left to right	any of the order	depends on compiler			left to right
Expression	173458	162^30	32^30	49152			49152

UNIT-IV

Questions	Choice 1	Choice 2	Choice 3	Choice 4	Answer
Each pass through a loop is called a/an	enumeration	iteration	culmination	pass through	iteration
Which looping process checks the test condition at the end of the loop?	for	while	do-while	no looping process checks the test condition at the end	do-while

A continue statement causes execution to skip to	the end of the program	the first statement after the loop	the statement following the continue statement	the next iteration of the loop	
In a group of nested loops, which loop is executed the most number of times?	the outermost loop	the innermost loop	all loops are executed the same number of times	cannot be determined without knowing the size of the loops	the innermost loop
The statement i++; is equivalent to	i = i + i;	i = i + 1;	i = i - 1;	i - - ;	i = i + 1;
Which looping process is best used when the number of iterations is known?	for	while	do-while	all looping processes require that the iterations be known	for
What's wrong? for (int k = 2, k <= 12, k++)	the increment should always be ++k	the variable must always be the letter i when using a for loop	there should be a semicolon at the end of the statement	the commas should be semicolons	
What's wrong? while((i < 10) && (i > 24))	the logical operator && cannot be used in a test condition	the while loop is an exit-condition loop	the test condition is always false	the test condition is always true	the test condition is always false
If there is more than one statement in the block of a for loop, which of the following must be placed at the beginning and the ending of the loop block?	parentheses ( )	French curly braces { }	brackets [ ]	arrows < >	French curly braces { }

What value is stored in num at the end of this looping? for (num = 1; num <= 5; num++)	1	4	5	6	4
Which of these selection statements test only for equality?	if	switch	if & switch	None of the mentioned	switch
Which of these are selection statements in Java?	if()	for()	continue	break	if()
Which of the following loops will execute the body of loop even when condition controlling the loop is initially false?	do-while	while	for	None of the mentioned	do-while
Which of these jump statements can skip processing remainder of code in its body for a particular iteration?	break	return	exit	continue	continue
Which of these statement is correct?	switch statement is more efficient than a set of nested ifs.	two case constants in the same switch can have identical values.	switch statement can only test for equality, whereas if statement can evaluate any type of boolean expression.	it is possible to create a nested switch statements.	two case constants in the same switch can have identical values.

Which of the following is one kind of a branching statement?	switch statement	break statement	compound statement	for statement	break statement
Which branching statement will cause a program to immediately exit a loop?	break	continue	return	All of the above	break
Which of the following branching statements is most appropriate for a java method?	break	continue	return	for	for
Which of the following branching statements is used to escape current execution (iteration) and transfer control back to the start of the loop?	break	continue	return	All of the above	continue
Which branching statement is used in a “switch” loop?	break	continue	return	None of the above.	break
A method in java that does not have a return value can possess any number of return statements?	FALSE	True	Depends	Cannot be determined	True
The statement that transfers control to the beginning of the loop is called	break statement	exit statement	continue statement	goto statement	continue statement

UNIT-V

Questions	Choice 1	Choice 2	Choice 3	Choice 4	Answer
What is the return type of a method that does not returns any value?	int	float	void	double	void
What is the process of defining more than one method in a class differentiated by method signature?	Function overriding	Function overloading	Function doubling	None of the mentioned	Function overloading
Which of the following is a method having same name as that of it's class?	finalize	delete	class	constructor	constructor
Which method can be defined only once in a program?	main method	finalize method	static method	private method	main method
Which of these statement is incorrect?	All object of a class are allotted memory for the all the variables defined in the class.	If a function is defined public it can be accessed by object of other class by inheritance.	main() method must be made public.	All object of a class are allotted memory for the methods defined in the class.	All object of a class are allotted memory for the methods defined in the class.
Which of these class is super class of every class in Java?	String class	Object class	Abstract class	ArrayList class	Object class
Which of these method of Object class can clone an object?	Objectcopy()	copy()	Object clone()	clone()	Object clone()
Which of these method of Object class is used to obtain class of an object at run time?	get()	void getclass()	Class getclass()	None of the mentioned	Class getclass()
Which of these keywords can be used to prevent inheritance of a class?	super	constant	Class	final	final
Which of these keywords cannot be used for a class which has been declared final?	abstract	extends	abstract and extends	None of the mentioned	abstract
Which of these class relies upon its subclasses for complete implementation of its methods?	Object class	abstract class	ArrayList class	None of the mentioned	abstract class
Which interface provides the capability to store objects using a key-value pair?	ava.util.Map	Java.util.Set	Java.util.List	Java.util.Collection	Java.util.Map

What is the output of this program?
class Output {
public static void main(String args[])
{
Object obj = new Object();
}
}
	object	class object	class.java.lang.object	compilation error	class.java.lang.object

What is the output of this program?
class Output {
static void main(String args[])
{
int x , y = 1;
x = 10;
if (x != 10 && x / 0 == 0)
 	
System.out.println(y);
else
 	
System.out.println(++y);
}
}
	1	2	Runtime Error	Compilation Error	Compilation Error

What is the output of this program?
class equality {
int x;
int y;
boolean isequal(){
return(x == y);
}
}
class Output {
public static void main(String args[])
{
equality obj = new equality();
obj.x = 5;
obj.y = 5;
 	
System.out.println(obj.isequal);
}
}
	FALSE	TRUE	0	1	TRUE

String in Java is a?	class	object	variable	character array	class
Which of these method of String class is used to obtain character at specified index?	char()	Charat()	charat()	charAt()	charAt()
Which of these keywords is used to refer to member of base class from a sub class?	upper	super	this	None of the mentioned	super
Which of these method of String class can be used to test to strings for equality?	isequal()	isequals()	equal()	equals()	equals()
Which of the following statements are incorrect?	String is a class	Strings in java are mutable	Every string is an object of class String	Java defines a peer class of String, called StringBuffer, which allows string to be altered.	Strings in java are mutable

Which of these function is used to allocate memory to array variable in Java?	malloc	alloc	new	calloc	new
Which of these is necessary to specify at time of array initialization?	Row	Column	Both Row and Column	None of the mentioned	row
What will be printed using following code block? int[] a = {0,1,2,3,4,5,6,7}; System.out.println(a.length);	6	7	8	9	8
Given a one dimensional array arr, what is the correct way of getting the number of elements in arr.Select the one correct answer.	arr.length	arr.length - 1	arr.size	arr.size - 1	arr.length
Arrays in Java are implemented as?	class	object	variable	None of the mentioned	object
Which of the following statements are valid array declaration ? (A) int number(); (B) float average[]; (C) double[] marks; (D) counter int[];	(A)	(A) & (C)	(B) & (C)	(D)	(B) & (C)
int number[] = new int[5]; After execution of this statement, which of the following are true? (A) number[0] is undefined (B) number[5] is undefined (C) number[4] is null (D) number[2] is 0 (E) number.length() is 5	(A) & (E)	(C) & (E)	(E)	(B), (D) & (E)	(B), (D) & (E)
Which one of the following will declare an array and initialize it with five numbers.	Array a = new Array(5);	int [] a = {23,22,21,20, 19};	int a [] = new int[5];	int [5] array;	int [] a = {23,22,21,20,19};
Which three are legal array declarations? 1.int [] myScores []; 2. char [] myChars; 3. int [6] myScores; 4. Dog myDogs []; 5. Dog myDogs [7];	1, 2, 4	2, 4, 5	2, 3, 4	All are correct.	1, 2, 4
Which of the following statements are valid array declarations? A. int number(); B. int number[]; C. double[] marks; D. counter int[];	A,B,C	B,C	C,D	A,B,D	B,C