Note Junction

Best Note Provider

Note By: Roshan BiSt

Unit-1 [Introduction to OOP]

Additional type defn

1> Characteristics / Terms of Object Oriented Languages:

⊛ Object-oriented programming → It is a programming language based on the concept of objects and classes which contain data in the form of fields often known as attributes and code in the form of procedures often known as methods.

1> Objects → Objects are variable of member function of a class which are user-defined data types. Objects are basic runtime entities in a object-oriented programming.
"p1" is the object in the example class below:

```
Class person {  char name[20];
                int id;
                public:
                    void getdetails(){ }
                };
int main(){
            person p1; // p1 is object of type
            }
```

Programming problem is analysed in-terms of objects and the nature of communication between them. Program objects should be choosen such that they match closely will the real world objects. Objects take up space in memory and have an associated address like structure or union in C. In fact objects are the variables of type class.

2> Class → A class is a collection of objects of similar type which are also user-defined data types and behaves like a built-in types of a programming language. Once a class has been defined, we can create any number of objects belonging to that class. The syntax to create an object is similar to the syntax used to create variable in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object mango belonging to class fruit.

Syntax:

```
class class_name {  access specifier:
                    // data members & member functions
                    };
```

We will use private, public and protected as access specifiers & no. of data members and member functions as our need in program.

iii) Data abstraction (or data hiding) and Encapsulation:

The wrapping up or combining of data and functions into a single unit is known as encapsulation. Those functions which are wrapping in the class can access it but the data are not accessible to outside world. This type of insulation of data from direct access is called data hiding or information hiding.

Data abstraction refers to providing only needed information to the outside world and hiding implementation details. The advantage of abstraction is we can change implementation at any point without affecting users of complex class.

iv) Inheritance → Inheritance is the process by which the objects of one class acquire the properties of objects of another class. It helps to share common characteristics with the class from which it is derived.

The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

For example: The bird 'Robin' is a part of class 'flying bird' which is again a part of class 'bird'.

v) Polymorphism (Overloading) → Polymorphism is the ability to make more than one form. Any operation may show different behaviours in different instances.

For example: Consider the operation of addition. For two numbers, the operation will generate sum but if the operands are strings, then the operation will produce third string by concatenation.

The process of making an operator to show different behaviours in different instances is known as operator overloading.

Unit→2 [Basics of C++ programming]

1) Explain the purpose of namespace with suitable example.

Ans: Namespace is a new concept introduced by ANSI C++ standards comittee. Namespace defines a scope for the identifiers that are used in a program. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when our code base includes multiple libraries.

Example:

```
#include<iostream>
using namespace std;
namespace first {
        void showinfo () {
        cout <<"Inside first" <<endl;
        }
namespace second {
        void showinfo () {
        cout <<"Inside second" <<endl;
        }
int main () {
        first:: showinfo. ();
        second:: showinfo ();
        return 0;
}
```

In the above program namespace helps to differentiate same function showinfo used in two differenb libraries. If we are not using namespace in above program then the compiler has no way of knowing which version of showinfo() we are refering to within our code.

2) Manipulators:

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

endl → The endl manipulator works same as the "\n" (i.e. newline) as we used in C. "\n" is also compatable in c++ too. The endl manipulator when used in an output statement causes to print in next line.

For example :-

```
cout<< "m=1" <<endl;
cout<< "n=2" <<endl;
```

This endl in first line of above example will cause to print m=1 in one line and n=2 in next line.

setw → The setw manipulator helps to shift the output statements to rightwards according to the need. We should include <iomanip> preprocessor file while using manipulators like setw in our program.

For example:-

```
cout << setw(50) << sum << endl;
```

The manipulator setw(50) in above example will right shift the output value of sum in the screen.

3) Function overloading with example:

Two or more functions having same name but different argument(s) are known as overloaded functions and this type of mechanism used in program is called function overloading.

In C++ programming, two functions can have same name if number or type of arguments passed are different. This is the concept of function overloading.

For example:

```
#include<iostream>
using namespace std;
float area(int);
int area(int,int);
float area(int, double);
int main(){
cout<<area(5)<<endl;
cout<<area(5,4)<<endl;
cout<<area(5,3.5)<<endl;
}
float area(int r){
return(3.14*r*r);
}
int area(int l,int b){
return(l*b);
}
float area(int b, double h){
return(0.5*b*h);
}
```

This is the program that overloads three functions with same name area but different return type and no. of arguments.

4.> Scope Resolution Operator (::)

We know that same variable name can be used to have different meanings in different blocks. A variable declared outside class or block is global variable and a variable declared inside class is local variable. In C the global version of a variable cannot be accessed from inner block. C++ resolves this problem by introducing new operate :: called scope resolution operator. This can be used to uncover hidden variable. It takes following form:

```
:: variable_name
```

The scope resolution operator basically does following two things

i) Access the global variable when we have a same local variable with same name.

ii) Defining a function outside the class.

5) Inline Function:

⊛ What is inline function? Where it is used? Illustrate with example.

Ans:- The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be made inline so that compiler can replace those function defination wherever those are being called.

It is only used where there are only small functions. It is not applicable for big functions. If function is big then the compiler can ignore the "inline" request and treat the function as normal function.

To make any function as inline, we start its definition with the keyword "inline".

Syntax

```
inline return_type function_name (parameter(s)) {
    //function code
}
```

The following program demonstrates the use of inline function.

```
#include<iostream>
using namespace std;
inline int cube (int s) {
    return s*s*s;
}
int main() {
    cout << "The cube of 3 is" << cube(3);
    return 0;
}
```

7) Dynamic memory allocation with new and delete:

(*) What are new and delete? Explain their use with suitable example.

Ans:- The process of allocating memory during execution time/runtime of program using heap space of memory to reduce wastage of memory is called dynamic memory allocation. New and delete are the keywords that help to allocate memory dynamically in our program. They are described below with example.

new operator → The new operator denotes a request for memory allocation on the heap. If sufficient memory is available, new operator initilizes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax:

```
pointer_variable = new data_type;
```

Example:

```
int *p = null;  // pointer initialized with null.
p = new int;    // Then request memory for variable.
```

delete operator → Since it is programmers responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax: delete pointer_variable;

Examples:

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

6 > Call by refrence / Pass by refrence:

⊛ What is the principle reason for passing arguments by refrence? Explain with code.

Ans: There may arise a situation where we would like to change the values of variables in the calling program using function which is not possible if call-by-value method is used. This is because the function does not have access to the actual variables in the calling program and can only work on the copy of the values. So, in such a case we need to pass value by refrence. Call-by-value method is fine if the function does not need to alter the values of the original variables in calling program.

Example for call by refrence:

```
#include<iostream>
using namespace std;
void change(int*);

int main(){
    int a=15;
    clrscr();
cout<<"Before calling function a="<<a;endl;
    change(&a); //passing arguments by refrence
cout<<"After calling function a="<<a;endl;
    getch();
}
void change(int*x){
    *x=*x+5;
}
```

Output:

Before calling function a=15
After calling function a=20.

If we passed argument by value in above program the value would not be altered from 15 to 20.

⊛ What is function overloading? Explain with example.

Ans: A function that share the same name are said to be overloaded functions and the process is referred to as function overloading. i.e function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters or different sequence of parameters or different numbers of parameter.

Example:

```
#include <iostream.h>
    float perimeter (float);
    int perimeter (int,int);
    int perimeter (int,int,int);
int main(){
    cout<<"Perimeter of a circle:"<<perimeter(2.0)<<endl;
    cout<<"Perimeter of a rectangle:"<<perimeter(10,10)<<endl;
    cout<<"Perimeter of a triangle:"<<perimeter(5,10,15);
    return 0;
}

float perimeter(float r){
                return (2*3.14*r);
                }

int perimeter(int l, int b){
                return (2*(l+b));
                }

int perimeter(int a, int b, int c){
                return (a+b+c);
                }
```

Unit-3 [Classes and Objects]

1) Specifying a class / & Specifying object:

The class declaration is similar to a structure declaration. The keyword class is used during declaration of class. A class is a way to bind data members and member functions together under some access specifier. The general form of class declaration is:

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

Once a class has been declared, we can create variable of that type by using the class name. In C++ the class variables are known as objects. Object can be created moreover in above program as follows in the main function:

```
int main(){
    class_name object1, object2... objectn;
}
```

2) Access Specifiers / Access Modifiers in C++:

Access specifiers are used to implement an important feature of object oriented programming known as data hiding. There are 3 types of access specifiers in C++ which are as follows:-

1) Public → All the class members (i.e, data members and member functions) are accessible from anywhere outside the class but within a program. We can set and get the value of public variables without any member.

⁂ Private → A private member variable or function cannot be accessed or even viewed from outside the class. Only the member functions of that class and friend functions can access private members.

⁂ Protected → A protected member variable or function is very similar to a private member but it is provided one additional benefit that they can be accessed in child classes which are called derived classes. Protected data can be accessed with the help of member functions of derived class.

3) Constructor:

⊛. What is constructor? Describe it with different types. Justify overloaded constructor with suitable example.

Ans: A constructor is a member function of a class which initializes objects of a class. In C++, constructor is automatically ~~of~~ called when object is created. It does not have any return type at all, not even void.

There are three types of constructors which are described below with examples.

1) Default Constructors → Default constructor is a constructor which does not take any argument (i.e, it has no parameters).

Example

```
#include <iostream>
using namespace std;
class construct { public:
    int a, b;
    construct() {
        a=20;
        b=10;
    }
};
```

II) Parameterized Constructors → It is possible to pass arguments to constructors. The constructors in which arguments are passed as parameters is called parameterized constructor. When we define the constructors body we use the parameters to initilize the object.

Example:

```
Class Point { private:
                int x, y;
              public:
                Point (int x1, int y1){
                                        x= x1;
                                        y=y1;
                                      }
                int getX() {
                                        return x;
                                      }
                int getY() {
                                        return y;
                                      }
};
```

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

```
class_name (const class_name &old_obj);
```

Example:

```
Class Point { private:
                int x, y;
              public:
                Point (int x1, int y1){
                                        x=x1;
                                        y=y1;
                                      }
Point (const Point &p2){
                                        x=p2.x;
                                        y=p2.y;
                                      }
  int getX() { return x; }
  int getY() { return y; }
};
```

Overloaded constructor with example:

If more than one constructor is defined within a class then it is called overloaded constructor or constructor overloading. It is similar to function overloading. Overloaded constructors essentially have same name as name of the class but difference in arguments passed to them.

Example:

```
#include<iostream>
using namespace std;
class abc{ private:
    int x,y;
public:
    abc() //Constructor 1 with no arguments
    {
        x=y=0;
    }
abc(int a) //Constructor 2 with one argument.
    {
        x=y=a;
    }
abc(int a, int b) //Constructor 3 with two arguments.
    {
        x=a;
        y=b;
    }
void display(){
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
};
int main(){ abc p; // constructor 1 called automatically
    abc q(10); // Constructor 2 called automatically
    abc r(10,20); //Constructor 3 called automatically
    p.display();
    q.display();
    r.display();
    return 0;
}
```

5) Destructor

Destructor is a member function which destructs or deletes an object. It is a member function which is used to destroy object that have been created by constructor. It has also same name as the class but is preceded by a symbol ~ (tilde).

(*) When is destructor called?

⇒ A destructor function is called automatically when the object goes out of the scope:
i) the function ends.
ii) the program ends
iii) a block containing local variables ends.
iv) a delete operator is called.

Example:

```
#include <iostream.h>
#include <conio.h>
class example { int a;
        public:
          example (){ a=0;
            cout <<"\n Inside the constructor";
                    }
          ~example (){
            cout <<"a=" <<a;
           cout <<"\n Inside the destructor"
                    }
};
int main (){ example e;
          cout << "Hello";
          return 0;
}
```

4) Friend function:

Friend function is a special function in object oriented programming which is used to get access the private and protected members of class from outside the class. The compiler knows a given function is friend function by the use of keyword friend. The declaration of a friend function should be made inside the body of the class starting with keyword friend.

Declaration

```
class class_name {
... ... ...
friend return_type function_name (argument(s));
... ... ...
};
```

Definition → No friend keyword is used in definition.

```
return_type function_name (argument(s)) {
... ... ...
/* Private and protected data of class name can be
   accessed from this function */
... ... ...
}
```

Example:

```
#include <iostream>
using namespace std;
class Box {
    double width;
  public:
    friend void printwidth (Box. box);
    void setwidth (double wid);
};
void Box :: setwidth (double wid.) {
    width = wid;
}
```

Not a member function of class

```
void printwidth (Box box) {
    cout << "Width of box: " << box.width;
}
int main () {
    Box box;
    box.setwidth (10.0);
    printwidth (box); //using friend function to
                      print width.
    return 0;
}
```

⊛. What is static data member illustrate with example.

OR

⊛. Create a real scenario where static data members are used.

Ans:- When we declare a normal variable (data member) in a class, different copies of those data members create with the associated objects.

In some cases when we need a common data member that should be same for all objects, we can not do this using normal data members. To fulfill such cases, we need static data members.

Defⁿ→ It is a variable which is ~~used~~ declared with the static keyword, it is also known as class member, thus only single copy of the variable creates for all objects.

Example:

```
#include<iostream>
using namespace std;
class demo {
    private:
        static int x;
    public:
        static void fun() {
            cout<<"Value of x:"<<x<<endl;
        }
};
int demo::x=10; //defining
int main(){
    demo x;
    x.fun();
    return 0;
}
```

Unit-4 [Operator Overloading]

#Explain Operator Overloading:

(1) The method of making operators to work for user-defined classes and having the ability to provide operators with a special user defined meaning is known as operator overloading.

For example → '+' operator can be overloaded to perform on various data types, like addition for integer, concatenation for string etc.

We can overload all the C++ operators except the following:

i) Class member access operators (., *).
ii) Scope resolution operator (::).
iii) Size operator (sizeof).
iv) Conditional operator (? :).

Syntax for defining opesator overloading.

keyword

```
return_type operator operator_symbol(argument list)
                {
                  function body // task defined
                }.
```

 # Program that overloads insertion (<<) and extraction (>>) operators.

In C++, insertion operator "<<" is used for output and extraction operator ">>" is used for input. We must know following things before we start overloading these operators.

i) cout is an object of ostream class and cin is an object of fstream class.
ii) These operators must be overloaded as a global function and if we want to allow them to access private data members of class, we must take them friend.

Example

```
#include<iostream>
using namespace std;
class Complex { private:
                  int real, imag;
                public:
                  Complex(int r=0, int i=0){
                    real=r; imag=i; }
friend ostream &operator<<(ostream &out, const Complex &c);
friend istream &operator>>(istream &in, Complex &c);
};
```

```
ostream &operator<<(ostream &out, const Complex &c)
        {
          out<< c.real;
          out<<"+i"<<c.imag<<endl;
          return out;
        }

istream &operator>>( istream &in, Complex &c)
        {
          cout<<"Enter real part";
          in>>c.real;
          cout<<"Enter imaginary part";
          in>> c.imag;
          return in;
        }

int main(){
          Complex c1;
          cin>>c1;
          cout<<"The complex object is";
          cout<<c1;
          return 0;
        }
```

Output:

Enter real part 10
Enter imaginary part 20
The complex object is 10+i20.

2) Unary Operator Overloading:

If overloading takes place on a single operand using unary operators like increament (++), decreament (--), unary minus (-), logical not (!) etc is called unary operator overloading.

The unary operators operate on the object for which they were called. Normally, this operator appears on the left side of object called prefix but sometimes they can be used as postfix as well.

Example: Program to understand unary operator overloading.
(OR Arithmetic minus (-) operator overloading)

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

Class space {
        int x, y, z;
    public:
        void getdata(int a, int b, int c);
        void display();
        void operator -(); //unary minus overloaded.
};

void space::getdata(int a, int b, int c){
        x=a;
        y=b;
        z=c;
}

void space::display(){
        cout<<x<<endl;
        cout<<y<<endl;
        cout<<z<<endl;
}

void space::operator -(){
        x=-x;
        y=-y;
        z=-z;
}

int main(){
        space s;
        s.getdata(10,-20,30);
        cout<<"S:";
        s.display();
        -s; //activates operator -() function.
        cout<<"-S:";
        s.display();
        return 0;
}
```

Output

```
S: 10
   -20
    30

-S: -10
    20
   -30
```

invoke → call

2) Binary Operator Overloading:

If overloading is taking place on two operands using binary operators is called binary operator overloading. The same mechanism is used to overload binary operator as in unary operator.

Example: Program to understand binary operator overloading.

```
#include<iostream>
using namespace std;
class complex {
    float x; // real part
    float y; // imaginary part.
public:
    complex(){ } //constructor 1
    complex(float real, float imag) //constructor 2
    {
        x = real;
        y = imag;
    }
    complex operator + (complex );
    void display ();
};
complex complex :: operator + (complex c) {
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
}
void complex::display(){
    cout << x << "+i" << y << endl;
}
int main() {
    complex C1, C2, C3; //invokes constructor 1
    C1 = complex (2.5, 3.5); //invokes constructor 2
    C2 = complex (1.6, 2.7);
    C3 = C1 + C2; //invokes operator +() function
    cout << "C1="; C1.display();
    cout << "C2="; C2.display();
    cout << "C3="; C3.display();
    return 0;
}
```

(return data type) (class name)

Output

```
C1 = 2.5+i3.5
C2 = 1.6+i2.5
C3 = 4.1+i6.2
```

Using Friend function:

If we want to overload previous discussed program using friend function then, we can modify as follows:

i) Replace the member function declaration by the friend function declaration.

ii) Redine the operator function as follows:

```
complex operator + (complex a, complex b) {
        return complex ((a.x+b.x), (a.y+b.y));
}
```

In this case, the statement C3=C1+C2; is equivalent to C3 = operator + (C1, C2);

3) Type Conversion:

Summary (OR Hierarchy of Stream Classes)

casting operator ⇔ conversion function

Conversion required	Conversion take place in	
	Source class	Destination class
Basic → Class	Not applicable	Constructor.
Class → Basic	Casting operator	Not applicable
Class → Class	Casting operator	Constructor.

Basic ⇔ Built-in
Class ⇔ User-defined. > Two data types in programming.

If the conversion takes place within the basic data types (i.e, built-in) data types then this type of conversion is called implicit conversion. or automatic conversion which is automatically done by the compiler. If the conversion takes place between uncompatible types as in the summary above then this type of conversion is called explicit conversion or also called type casting.

source — destination

i) Conversion from basic (built-in) type to class (user-defined) type:

The conversion from basic type to class type (i.e, built-in data type to user defined data type) can be illustrated with the following example.

```
#include<iostream.h>
using namespace std;
  class time { int year;
               int month;
               public:
                    time (int y, int m) { year = y;
                                          month = m;
                                        }
```

use of constructor

```
                    time (float a) { year = int(a)
                                     month = 12*(a-year);
                                   }

               void display () { cout<<"year = "<<year<<endl;
                                 cout<<"month = "<<month;
                               }
  };
int main(){ float y;
            cout<<"Enter the year";
            cin>>y;
            time t = y;
            t.display();
          }
```

obj

class type data

1) Conversion from class (user defined) to basic (built-in) type:

source

destination

We need parameterized constructor for converting basic type to class type in previous section but construction functions does not support this type of operation in this type of conversion. So, C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of overloaded casting operator is as follows:

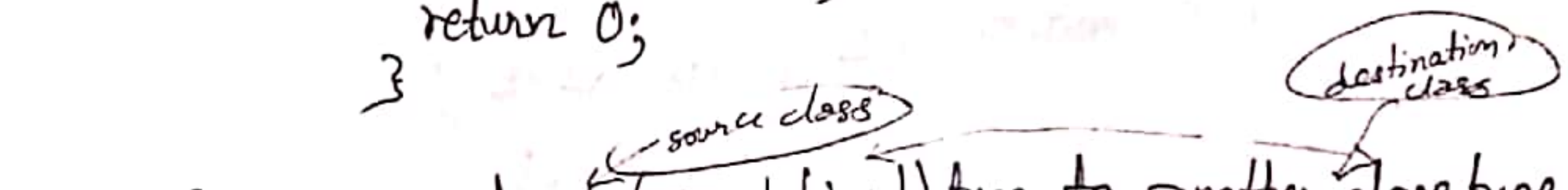
```
operator type_name(){
    ....  (function statements).
}
```

Example

```
#include<iostream>
using namespace std;
class time { int hrs;
             int mins;
             public:
             time (int h , int m){
                                  hrs=h;
                                  mins=m;
                                  }
        void display (){
                        cout <<hrs<<"hours",
                        cout <<mins <<"minutes";
                                  }
operator int() {return hrs*60+mins;
                }

};

int main(){ int duration;
            time t(3,20);
            t.display();
            duration=t;
            cout <<duration;
            return 0;
          }
```



ppt) Conversion from one class (user defined) type to another class type.

One class type can be converted to another class type using conversion function in source class and using construction function in destination class. In case of conversion between objects constructor function is applied to destination class. A conversion function is applied to source class.

Example

```
#include<iostream>
using namespace std;
class rupee {public:
                int rs;
                void show(){
                        cout<<"money in rupees"<<rs;
                        }
};
class dollar {public:
                int doll;
                dollar(int x){doll=x;
                        }
                operator rupee(){rupee temp;
                temp.rs=doll*50;
                return temp;
                }
                void show(){
                        cout<<"money in dollars"<<doll;
                        }
};
int main(){ dollar d1(5);
            d1.show();
            rupee r1=d1;
            r1.show();
            return 0;
}
```

Unit→5 [Inheritance]

⊛. What is the role of protected access specifier in inheritance? Explain with example.

Ans: If protected access specifier is used while deriving class then the protected member of the base class becomes the protected member of the derived class and private members of the base class are inaccessible.

Simply when a base class is inherited with protected by derived class then it can be accessed within class and from any class derived from this class. These members can't be accessed from outside of these (same as in derived class) and hence help to achieve the concept of data hiding.

Example: Program to demonstrate protected access specifier.

```
#include<iostream.h>
using namespace std;
class base {  private:
                int x
              protected:
                int y
              public:
                int z;
              base (){  x=1;
                        y=2;
                        z=3;
                     }
};
class derived: protected base {public:
                                 void showdata () {
cout<<"x is not accessible"<<endl;
cout<<"Value of y is"<<y<<endl;
cout<<"Value of z is"<<z<<endl; }
                               };
```

```
int main() {derived a;
            a.showdata();
            return 0;
           }
```

⊛. Differences between function overloading and function overriding

Function overloading	Function overriding
i) No keyword is applied during overloading.	i) Function which is to be overridden is preceded by keyword 'virtual' in the base class.
ii) Overloading can occur without inheritance.	ii) Overriding of functions occur when one class is inherited from another class
iii) Overloading a method in C++ is the ability for functions of the same name to be defined as long as these methods have different set of parameters.	iii) Method overriding is the ability of the inherited class rewriting the virtual method of the base class.
iv) Destructor can not be overloaded.	iv) Destructor can be overridden.
v) Overloading achieves early binding.	v) Overriding refers to late binding.
vi) Time of accomplishment is compile time.	vi) Time of accomplishment is run time.
vii) Overloaded function are in same scope	vii) Overridden function are in different scope.

⊛. What is abstract base class? Give an example.

Ans:- An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (i.e, to be inherited by other classes). It is a design concept in a program development and provides a base upon which other classes may be built. The classes student in the example is abstract classes since it is not used to create any objects.

```
#include<iostream>
using namespace std;
class student {
        int x;
      public:
        virtual void fun()=0; //pure virtual function.
        int getX {
                return x;
        }
};

class derived: public student {
                int y;
      public:
        void fun(){
cout<<" derived class function called ";}
};

int main() { derived d;
d.fun();
return 0;
}
```

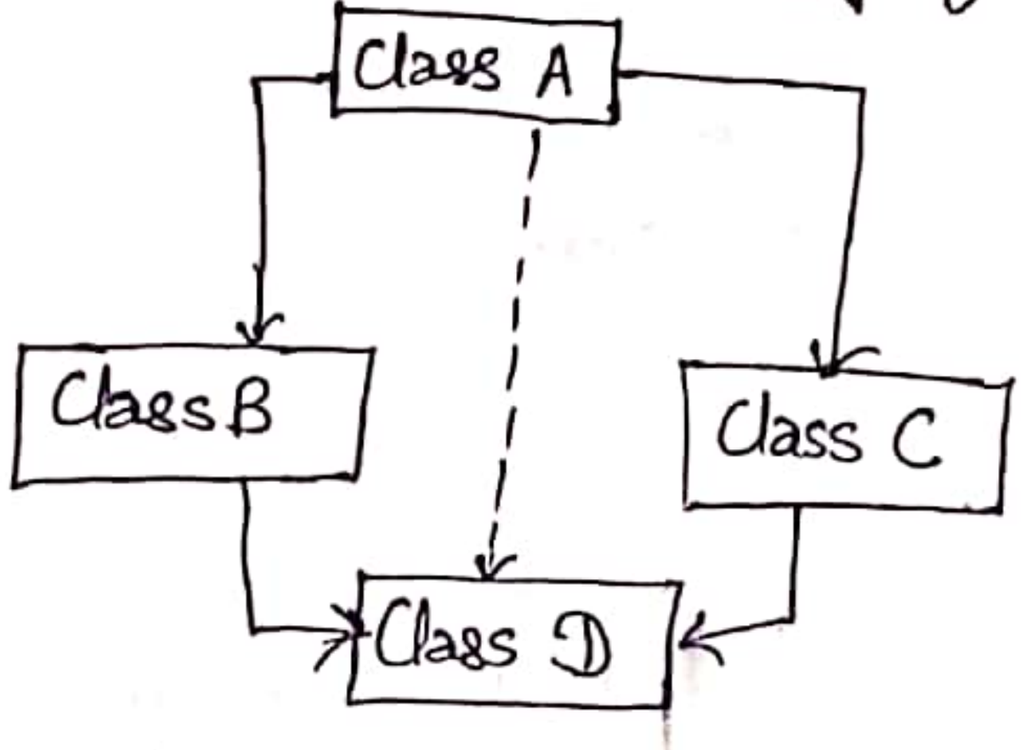
Abstract base class (Def^n) → Next def^n

If any class consists of the pure virtual function such class is known as abstract base class.

⊛. Define the various ambiguity situations that may occur during the process of inheritance. How can you resolve the ambiguity situation?

Ans: [OR Ambiguity in multiple Inheritance]

Ambiguity in C++ occur when a derived class have two base classes and these two base classes have one common base class. Consider the following figure.



In this figure, both class B and class C inherit class A, they both have single copy of class A. However class D inherit both class B and class C, therefore class D have two copies of class A, one from class B and another from class C.

If we need to access the data member of a class A through the object of class D, we must specify the path from which data member (let a) will be accessed, whether it is from class B or class C, because compiler can not differentiate two copies of class A in class D. There are two ways to avoid C++ ambiguity. One is using virtual base class as we use in hybrid inheritance and another using scope resolution operator. Using scope resolution operator we can manually specify the path from which data member (let a) will be accessed as in statements below:

```
obj. classB :: a = 10;
obj. classC :: a = 100;
```

Note → For long questions from inheritance understand the programs of types of inheritance with private and protected access specifier

⊛ Differentiate between base class and derived class with examples.

Ans:-

Base class (Parent class)	Derived class (Child class)
i) Base class helps to derive or create new classes.	i) Derived class is created from an already existing class.
ii) Base class cannot inherit properties and methods of the derived class.	ii) Derived class can inherit the properties and methods of the base class.
iii) It is also called parent class or superclass.	iii) It is also called child class or subclass.

iv) Syntax
class BaseClass {
//members...
}
	iv) Syntax
	class DerivedClass : access specifier BaseClass {
	//members...
	}

v) The base class members are contained by the object of derived class.	v) The derived class can add new members or change base class members.
vi) The changes done to base class will affect the derived class	vi) The changes done to derived class will not affect the base class.

⊛. Differentiate between class template and function template.

Ans.

Class template	Function template.
i) A class template defines a family of classes.	i) Function templates are special functions that can operate with generic (comprehensive) types..

ii) Syntax:
template <class T>
class add { }; //class template add
	ii) Syntax:
	template <class T>
	returntype function_name (arguments of type T){
	//Body of function with type T
	}

iii) The name of a template class is a compound name consisting of the template name and the full template argument list enclosed in angle braces.	iii) A function template has the name of the function template and the particular function chosen to resolve a given template.
iv) When instantiated, a class template becomes a class.	iv) When instantiated, a function template becomes a function.
v) Class templates can be used to write programs at compile-time.	v) Function templates can be used for creating factory functions.
vi) Class templates can be partially specialized.	vi) Function templates can't be partially specialized.
vii) Class templates can have default template parameters.	vii) Function templates can't have default template parameters.

⊛ Explain the concept behind macros? Write a program to find area of circle using macro.

Ans:- Macros are designed to reduce the function call overhead in case of small functions. The concept of macros are not unique to C++, they are also supported by C language. Macros are defined by using hash sign because they are processed by macro processors & compiler.

Example:

```
#include<iostream.h>
#include<conio.h>
#define Pi 3.1416  //macro.
int main(){
        float area, radius;
        clrscr();
        cout<<"Enter radius of circle"<<endl;
        cin>>radius;
        area = Pi*radius*radius;
        cout<<"Area of circle is:"<<area<<endl;
        return 0;
        }
```

⊛ What is exceptional handling? WAP in C++ that make use of exceptional handling.

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

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

Example:- Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

```
#include<iostream>
using namespace std;
int main(){
    int x=-1;
    cout << "Before try \n";
    try {
        cout <<"Inside try \n";
        if(x<0)
        {
            throw x;
            cout <<" After throw (Never executed) \n";
        }
    }
    catch(int x){
        cout<<" Exception caught \n";
    }
    cout <<"After catch (will be executed) \n";
    return 0;
}
```

Output:

```
Before try
Inside try
Exception caught
After catch (will be executed).
```

For more

Refer the full note for unit 6, 7, 8
mainly 7 and 8 for remaning note
they are shorter and all topics equally
important.

If my notes really helped you,then you can support me on esewa for my hardwork.

Esewa ID: 9806470952