29/9/2022

INTRODUCTION

★ Variables are stored in heap area of memory.

★ In-built data-types : 1. Integer
2. Float
3. Char

★ User-defined data-types : 1. Structures
2. Unions
3. Enums.

★ In C++, we do not use struct structname while declaring structure variables. Only use structname.

★ Class (User-defined data-type)

→ Can contain functions.

→ Needs access specifiers.

→ Class variable is called object. Object is instance to cla

→ Object help us to access class attributes (member variables, member functions).

→ class student
{
public : // allows access
int roll;
char name[20];
int addition()
{
int a=10, b=20, tot;
tot = a+b; return tot;
}
};

→ Class is a blueprint for objects. When we instanciate the class, object is created.

```
int main()
{
    student obj;
    obj.roll = 100;
    return 0;
}
```

★ Scope resolution operator to ~~dis~~ use function outside class = ::

★ Outside class:

→
```
class student
{
    int roll;    // private
    float percent; public:    // private (& percent)
    void get_data(); // public
    void display()    // public
    {
        ____________
        ____________
    }
};
void student:: get_data()  // Defining function outside class
{
    roll = 100;   // as it is declared in class.
}
int main()
{
    student obj;
    obj.get_data();
    obj.display(); return 0;
}
```

★ cout << "Enter roll number"; // printf
cin >> roll; // scanf

★ class student // For cin and cout.
{
int roll;
float percent;
public:
void get_data();
void display()
{
cout << "Roll number and percent are: ";
cout << roll << percent;
}
};
void student :: get_data()
{
cout << "Enter roll number and percentage :";
cin >> roll >> percent;
}
int main()
{
student obj;
obj.get_data();
obj.display();
return 0;
}

★ Objects are created by Constructor and are destroyed by Destructor. Name of the Constructor is the same as that of the class.

* Constructor helps us to initialise the objects. To initialise the values of attributes of the objects. Constructor is present in the public section of the class.

* Types of Constructors:
1. Default constructor
2. Parameterised constructor
3. Copy constructor

* class student // For default & parameterised constructor
{
    int roll;
    float per;
    public:
    void get_data();
    void display();
    student() // default constructor
    {
        roll=0;
        per=0;
    }
    student(int r, float p) //parameterised constructor
    {
        roll=r;
        per=p;
    }
};
int main() {
    student obj, obj1(10,80); // obj→default, obj1→parameterised
    obj.get_data(); return 0; }

```
class student()        // For destructor.
{
    int roll;
    float per;
    public:
        student(int r, int p)
        {
            roll=r;
            per=p;
        }
        student()
        {
            roll = 0;
            per = 0;
        }
        /*~student()          // destructor
        {
            cout<<"\n Bye Bye....";
        } */
        void getdata();
        void display();
        ~student()
        {
            cout<<"\n Bye Bye...";
        }
};
void student::getdata()
{
    cin>>roll>>per;
}
void student::display()
```

```
{
    cout<< "\n Your data: ";
    cout<<roll<<" "<<per;
}
int main()
{
    student obj1, obj2(10,89.90);
    obj1.getdata();
    obj1.display();
    obj2.display();
    return 0;
}
```

★
```
class student          // For copy constructor
{
    int roll;
    float per;
    public:
        student(int r, int p)
        {
            roll=r;
            per=p;
        }
        student(student &ob)    // copy constructor
        {
            roll=ob.roll;
            per=ob.per;
        }
        void getdata();
        void display();
```

```
    ~student()
    {
        cout<<"\nBye Bye....";
    }
};
void student::getdata()
{
    cin>>roll>>per;
}
void student::display()
{
    cout<<"\nYour data:";
    cout<<roll<<" "<<per;
}
int main()
{
    student obj1, obj2(10,89.90);
    student obj3=obj2;
    obj1.getdata();
    obj1.display();
    obj2.display();
    obj3.display();
    return 0;
}
```

★ Friend Member Function (Has to be used in the presence of obj)

→
```
class student
{
    int roll;
    float per;
    public:
```

```
        void getdata();
        friend void display(student);
};
void student :: getdata()
{
    cin>> roll >>per;
}
void display (student ob)
{
    cout<<ob.roll;      // or declare ob and access roll
    cout<<"\n"<<ob.per;
}
int main()
{
    student obj;
    obj.getdata();
    display(obj);
    return 0;
}
```

★ namespace creates logical partitions to handle different classes.

★ cout << endl; : New-line character.

★ Namespaces

→
```
#include <iostream>
using namespace std;
namespace one
{
    void display()
```

```
        {
            cout<<"Hi all 1"<<endl;
        }
}
namespace two
{
    void display()
    {
        cout<<"Hi all 2"<<endl;
    }
}
int main()
{
    one::display();    //prints "Hi all 1".
    two::display();    //prints "Hi all 2".
    return 0;
}
```

★ To use Global Variable Using Scope Resolution.

→
```
#include<iostream>
using namespace std;
int var = 10;
int main()
{
    int var;
    var = 100;
    cout<<"\n"<<var;    //prints 100
    cout<<"\n"<<::var;  //prints 10
    return 0;
}
```

☆ Class as Friend function:

→

```
#include <iostream>
using namespace std;
class teacher;   //forward referencing.
class student
{
    int roll;
    public:
        student()      //Constructor.
        {
            roll = 10;
        }
        friend class teacher;
};
class teacher  //Can access roll
{
    public:
        void display();
};
void teacher :: display()
{
    student ob;
    cout << ob.roll ;   //prints 10
    cout << "\nTeacher's display";
}
int main()
{
    teacher obj;
    obj.display();
    return 0;
}
```

✱ Inline Function (Request to the Compiler)

→
```
inline void display()
{
    cout<<"Hi";
}
int main()
{
    display(); //copies contents of display function (done by compiler) and executes with going to the function. Speeds up execution by saving time.
    return 0;
}
```

→ Only works on simple functions.

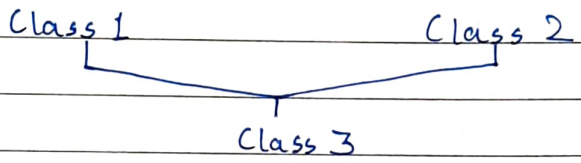
→ The compiler will not copy the lines if the function is complicated.

→ Saves from switching overhead.

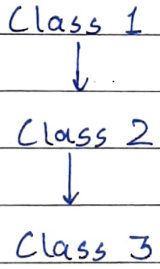
6/10/2022

INHERITANCE

- Parent class inherits the properties. (Base class)
- Child class gets the properties. (Derived class).
- Inheritance from both parents: Multiple inheritance.
- Inheritance from ~~from~~ parents and grandparents: Multi-level inheritance.
- Multiple Inheritance



- Multi-level Inheritance



- Access Specifiers:

1. Public.
2. Private.
3. Protected.

- When a derived class inherits publicly from the base class, it inherits only the public members, and protected members.
- Private members never get inherited.
- class Rect: public Shape

* When we publicly inherit properties of base class from inside the derived class, the protected properties remain protected and public properties remain public in the derived class.

* ```
#include <iostream>   // Public Inheritance.
using namespace std;
class Shape
{
    protected:
        int length;
        int breadth;
    public:
        void getdata(int l, int b)
        {
            length=l;
            breadth=b;
        }
};
class Rect : public Shape
{
};
int main()
{
    Rect obj;
    obj.length=100; //error as length is protected.
    obj.getdata(10,20); //works as the function is public
    return 0;
}
```

* Private and protected members behave the same,
```

except during inheritance. Private members get inherited but private members never get inherited.

★ If properties are inherited using protected method, then public members also become protected inside the derived class.

★ Protected Inheritance.

→
```
#include <iostream>
using namespace std;
class Shape
{
    protected:
        int length, breadth;
    public:
        void getdata()
        {
            cout << "Enter length and breadth: ";
            cin >> length >> breadth;
        } private: int area;
};
class Rect: protected Shape
{
};
int main()
{
    Rect obj;
    obj.getdata();      // will give an error.
    return 0;
}
```

* If members of a class are inherited using the private method, the public and protected member become private in the derived class.

→ class Rect: private Shape

* ~~Inheriting~~

* Inheriting Two Classes. [Multiple Inheritance].

→
```
#include<iostream>
using namespace std;
class Shape
{
    protected:
        int length, breadth;
    public:
        void getdata()
        {
            cout<<"Enter length and breadth: ";
            cin >> length >> breadth;
        }
};
class Area
{   //public:
    void cal_area()
    {
    }
};
class Rect: public Shape, Area
{
    //Nothing is inherited from class Area.
};
int main()
```

```
{
    Rect obj;
    obj.get_data();  //works
    obj.get obj.cal_area();  //will give an error.
    return 0;
}
```

→ Both inherited classes are publicly inherited.

→ If nothing is specified like:
class Rect: Shape, Area
The classes are inherited using private method by default.

→ Inheritance methods can be specified differently for different inherited classes.

Ex. class Rect: public Shape, private Area

★ Diamond Problem

→
```
#include<iostream>
using namespace std;
class T
{
    public:
        T()
        {
            cout<<"\nHello T.";
        }
};
class A: public T
{
    public:
        A()
        { cout<<"\n Hello A."; }
```

```
};
class B: public T
{
    public:
        B()
        { cout<<"\nHello B."; }
};
class C: public A, B
{
    public:
        C()
        { cout<<"\nHello C."; }
};
int main()
{
    C obj;
    return 0;
}
```

→ Output:

```
Hello T.
Hello A.
Hello T.
Hello B.
Hello C.
```

→ This is ~~ambiguity~~ ambiguity.

→ Solution:

```
#include<iostream>
using namespace std;
class T
{
    public:
```

```
        T()
        { cout<<"\nHello T."; }
};
class A: virtual public T      // prevents printing T twice
{
    public:
        A()
        { cout<<"\nHello A."; }
};
class B: virtual public T      //prevents printing T twice
{
    public:
        B()
        { cout<<"\nHello B."; }
};
class C: public A, B           //A is inherited before B. A forms
{                              // first. Then B.
    public:
        C()
        { cout<<"\nHello C."; }
};
int main()
{
    C obj;
    return 0;
}
```

→ Output:

```
Hello T.
Hello A.
Hello B.
Hello C.
```

→ Constructors always get executed from Top to Bottom and destructors always get executed from Bottom to Top.

→
```
#include<iostream>
using namespace std;
class T
{
    public:
        T()
        { cout<<"\nHello T."; }
        ~T()
        { cout<<"\nBye T."; }
};
class A: virtual public T
{
    public:
        A()
        { cout<<"\nHello A."; }
        ~A()
        { cout<<"\nBye A."; }
};
class B: virtual public T
{
    public:
        B()
        { cout<<"\nHello B."; }
        ~B()
        { cout<<"\nBye B."; }
};
class C: public A, B
{
```

```
public:
    C()
    { cout<<"\nHello C."; }
    ~C()
    { cout<<"\nBye C."; }
};
int main()
{
    C obj;
    return 0;
}
```

→ Output:

```
Hello T.
Hello A.
Hello B.
Hello C.
Bye C.
Bye B.
Bye A.
Bye T.
```

★ Types of Inheritance

1. Single-level inheritance

→ Public / protected members are inherited from class A to class B.

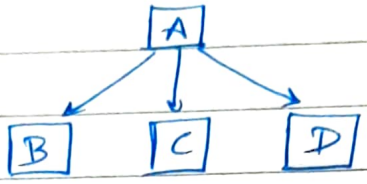
2. Multilevel inheritance

→ Members of class A is inherited into class B and then from B to class C.

3 Hierarchical inheritance

→ Members of class A is inherited by many other classes

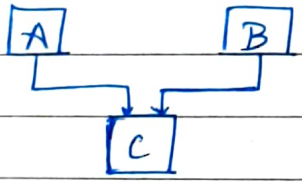
→



4 Multiple inheritance

→ Members of several classes are inherited by a single class.

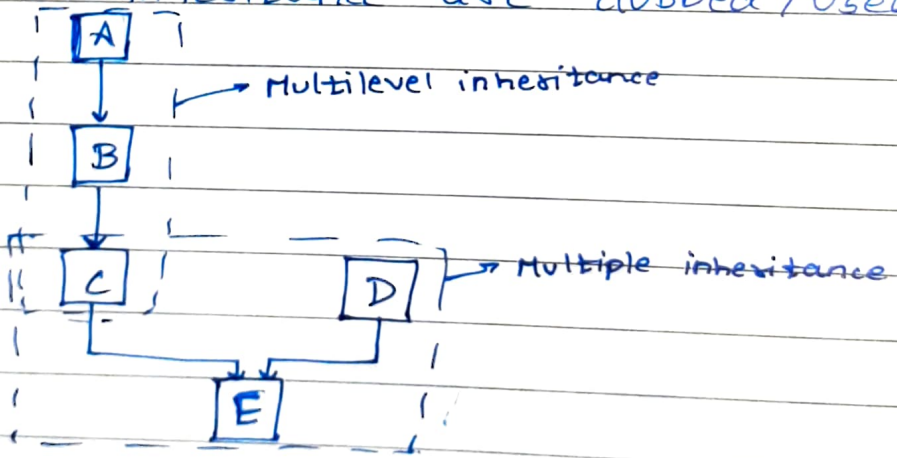
→



5 Hybrid inheritance.

→ Methods of inheritance are clubbed / used together

→



→ Combination of several inheritance methods.

—X—

2/10/2022

POLYMORPHISM

★ Function Overloading

→ Functions having the same name but different parameters, causing function overloading.

→
```
int addition (int a, int b)
{
    int total;
    total = a+b;
    return total;
}
float addition (float x, float y)
{
    float t = x+y;
    return t;
}
int main()
{
    int a=10, b=20;
    int tot = addition(a,b);
    cout<< tot <<endl;   //prints 30
    float x=10.23, y=20.23;
    float ftot = addition (x,y);
    cout<<ftot<<endl;   //prints 30.46
    tot = addition(x,y);
    cout<< tot <<endl;  //prints 30
    ftot = addition (a,b);
    cout<< ftot <<endl; //prints 30
    return 0;
}
```

Ex: ~~add~~ void add ();
void add (int x, int y);
void add (float p, float q, float r);

→ This is called "Compile time Polymorphism".

* Wrong Code

→
```
#include <iostream>
using namespace std;
class Complex
{
    int real, imag;
    public:
        complex(int x, int y)
        {
            real = x;
            imag = y;
        }
};
int main()
{
    Complex ob1(2,6), ob2(3,4);
    Complex ob3;
    ob3 = ob1 + ob2; //ob3.real = ob1.real + ob2.real;
    return 0;          //ob3.imag = ob1.imag + ob2.imag;
}
```

* Operator Overloading

→
```
#include <iostream>
using namespace std;
class Complex
{
    public:
        int real, img;  complex(){ };
        Complex (int x, int y)
```

```
    {
        real = x;
        img = y;
    }
    Complex operator + (Complex c)
    {
        Complex c3;
        c3.real = real + c.real;
        c3.img = img + c.img;
        return c3;
    }
};
int main()
{
    Complex c1(2,3), c2(5, 7), c3;
    c3 = c1 + c2;    // c1 will be invoked and c2 will be
    passed as its parameter. The result will be ~~set~~
    returned to c3. similar to c1.add(c2);.
    return 0;
}
```

```
#include<iostream>
using namespace std;
class A
{
    int integer1, integer2;
    public:
        A()
        {
            integer1=10;
            integer2=20;
        }
        void operator<<(A class_var)        //var <-> obj
        {
            cout<<integer1<<" "<<integer2;
        }
};
int main()
{
    A Renuka, class_var;
    Renuka<<class_var;
    return 0;
}
```

* Static Variables (Change in any one object applies to all).

→

```
#include<iostream>
using namespace std;
class Sample
{
    static int var;
    public:
        Sample()
```

```
    {}
    void getdata(int x)
    {
        var = x+10;
    }
    void display()
    {
        cout<<"Value of the variable is "<< var <<en
    }
};
int Sample:: var = 0;
int main()
{
    Sample obj, obj2;
    obj.getdata(1);
    obj.display();
    obj2.display();
    return 0;
}
```

→ Output:

```
Value of the variable is 11
Value of the variable is 11
```

★ Static Functions (★ only allow static variables).

→ Non-static functions can manipulate static variables.

→ Static functions will not change non-static variables and will throw an error.

→
```
#include<iostream>
using namespace std;
```
~~etc~~

```
class sample
{
    static int var;
    int nonstatic;
    public:
        sample()
        { }
        static void getdata (int x)
        {
            var = x + 10;
            nonstatic = 100;    //Only cause of error.
        }
        static void display()
        {
            cout<< "Value of the variable is "<< var<<endl;
        }
};
int sample::var = 0;
int main()
{
    sample obj, obj2;
    obj.getdata(1);
    obj.display();
    obj2.display();
    return 0;
}
```

★ Function Pointers:

→
```
int addition (int x, int y)
{
    return (x+y);
}
int main()
{
    int (*funptr) (int, int);
    funptr = addition;
    int sum;
    sum = funptr (10, 20);
    return 0;
}
```

★ Generic Pointer

→ It is a pointer which can be typecasted into any data type.

→ void *

→ It gives us the memory location.

→ Data type of pointer returned by malloc is void, wh is then typecasted to the desired datatype.

★ * : Indirection operator

→ : Dereference operator.

★ Pointers to Objects

→
```
class Sample
{
    public:
    virtual void display()
    {
```

```
        cout << "\n Display in Sample class.";
    }
};
class Derived : public Sample
{
    public:
        void display()
        {
            cout << "\n Display in Derived class.";
        }
};
int main()
{
    Sample * ptr;
    ptr -> display();  // display() of Sample class
    Derived obj;
    ptr = &obj;
    ptr -> display();  // display() of Derived class
    (*ptr).display();
    return 0;
}
```

→ Due to virtual function being used, compiler takes the decision to use Derived class' display function (for 2nd ptr) ~~when~~ during runtime.

→ Hence, this is Runtime Polymorphism.

→ If virtual keyword ~~is~~ was not used, the ptr (when used for the 2nd time) would use the base class' (Sample class) display() function.

→ # Without using 'virtual':

```
(*ptr).display();   // Displays "Sample class";
obj.display();      // Displays "Derived class";
```

* Classes having at least one function which has no definition (void display() = 0;) is called an abstract class. These have pure virtual functions.

* When constructors of a class is created in private section, we cannot create an object to that class.

* Objects to abstract classes cannot be created.

* If virtual functions from one class is inherited by another class (derived), an object to the derived class cannot be created either. In order to create an object, the functions are to be implemented. (Functions which are purely virtual)

* 'This' Pointer

→ To understand 'this' pointer, it is important to know how objects look at the functions and data members of a class.

1. Each object gets its own copy of the data members.
2. All access the same function definitions as present in the code segment.

Meaning, each object gets its own copy of data members and all objects share a single copy of member functions.

→ Then now, the question is that if only one copy of each member functions exists and is used by multiple objects, how are ^ member functions proper data members accessed and updated?

→ For this, compiler supplies an implic pointer

along with the names of the functions as 'this' pointer.

Ex:
```
class Sample
{
    int x;
    public:
        void display (int x)
        {
            this->x = x;        // this->x is the x declared in
        }                       // the class. (class member)
};
int main()
{
    Sample obj;
    obj.display(20);
    return 0;
}
```

* Generic Programming =

→
```
#include <iostream>
using namespace std;
template <typename T> // Generic data type
T addition (T x, T y)
{
    T tot = x+y;
    return tot;
}
int main()
{
    T sum;
    sum = add <int> (10,20);
```

```
    cout << sum << endl;    //30
    sum = add <float> (10.71, 20.61);
    cout << sum << endl;    //31.32
    return 0;
}
```

→ Generic data type can be typecasted to any other data type.

★ Function Template

→ For every function, there is a new template associated with it.

```
→ template <typename T>   // scope is designed only for 'display'.
void display (T a)
{
    cout << a;
}
template <typename X, typename Y>
void show (X b, Y c)
{
    .....
    .....
}
int main()
{
    display <int> (10);
    display <float> (10.7);
    show <int, char> (10, 'c');   //<float, int> or <double, int>
    or <char, float>, etc.
    return 0;
}
```

* Generic Programming:

→ It is a type of programming where type of data is specified at runtime.

→ We are writing a generalised code and compiler decides for what type of data it should work.

→ Template is the best example of generic programming in C++.

→ ~~Here~~ In generic programming, we are writing the algorithms which are applicable to a variety of data types.

→
```
#include <iostream>
using namespace std;
template <typename T>
class Sample
{
    T a, b;
    public:
        Sample (T x, T y)
        {
            a=x;
            b=y;
        }
        void add()
        { cout<<(a+b); }
};
int main()
{
    Sample <int> ob(10, 5);
    ob.add();
    return 0;
}
```

→
```
#include <iostream>
using namespace std;
template <class T1, class T2>
class Sample
{
    T1 a;
    T2 b;
public:
    Sample (T1 x, T2 y)
    {
        a=x;
        b=y;
    }
    void add()
    { cout<<(a+b); }
};
int main()
{
    Sample <int, float> ob(10, 3.5);
    ob.add();
    return 0;
}
```

* Exception Handling:

→
```
int main()
{
    int num1, num2;
    cout<<"Enter num1 and num2 : ";
    cin>>num1>>num2;
    int total = num1 + num2;
    int ans = num1 / num2;
```

```
    cout << "\n" << ans;
    return 0;
}
```

→ If num2 = 0, program gives us "Divide by 0" error.
→ Code will have 'try' block and 'catch' block.
→ Main purpose of exception handling is to identify and report the runtime error in the program.
→ Famous examples are:
- Divide by 0.
- Array index is out of bound.
- File not found
→ Exception handling is possible using three keywords:
- Try
- Catch
- Throw
→ Exception handling performs the following task:
- Find the problem in the given program. It is also called as Hit exception.
- When we receive the error information, then write a code to handle (or catch) the exception.
- In exception handling, corrective actions taken against raised exceptions is called exception handling.

→ Try
- It is a block of code in which there are chances of runtime error.
- This block is followed by one or more catch blocks.
- Most error-prone code is added in try block.

→ Catch
- This is used to catch the exception which is thrown by try block.
- In catch block, we take corrective actions on the

thrown exception:

→ Throw

- Program throws exception when a problem occurs.
- It is possible with 'throw' keyword.
- If we are using a statement like "throw (10)", then it will throw an exception with a value 10.

→ One try block can have ~~only~~ one, or more catch ~~blo~~ blocks

Ex.
```
int main()
{
    int num1, num2, ans;
    cout<<"\nEnter num1 and num2: ";
    cin>>num1>>num2;
    try
    { if(num2==0) { throw(num2); }
        ans = num1/num2;
    }
    catch (int exception)
    {
        cout<<"Divide by zero error is caught";
    }
    return 0;
} }
```

→ If num2 is zero, the try block "throws" num2 ~~to~~ (error exception is found) and the catch block rectifies it.

→ Try block should be immediately followed by catch block.

→ We cannot have statements between try and catch block.

→ Whenever try block throws an exception, program control leaves try block and enters into the catch block.

→ If an appropriate catch block is found, then the catch block is executed to handle the exception.

→ If no exception is detected and thrown, then the program will execute normally.

→
```
#include <iostream>
using namespace std;
int main()
{
    int arr[5], loc;
    cout<<"\nEnter 5 elements of array : ";
    for (int i=0; i<5; ++i)
        cin>>arr[i];
    cout<<"\nEnter location of element to retrieve: ";
    cin>>loc;
    try
    {
        if (loc>4 || loc<0)
            throw (loc);
        cout<<"The element is: "<<arr[loc];
    }
    catch (int location)
    {
        cout<<"\nArray index out of bound!\n";
    }
    return 0;
}
```

★ Throwing and Catching Exceptions:

→ Whenever an exception occurs, it is thrown by using the "throw" keyword.

→ "throw" keyword has different forms.

→ When we throw an exception, it will be caught by "catch" statements. It means that the program control will transfer from "try" to "catch" block.

→ Code for handling exceptions, is written in the "catch" block. Code for checking statements prone to exceptions is written in the "try" block.

→
```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "\n Enter an integer from 1 to 3: ";
    cin >> num;
    try
    {
        if (num == 1)
            throw (1);
        if (num == 2)
            throw (2.1);
        if (num == 3)
            throw ('a');
    } else
            throw(
    catch (int except)
    {
        cout << "\n You chose an integer.\n";
    }
    catch (float except)
    {
        cout << "\n You chose an float value.\n";
    }
```

```
    catch (char except)
    {
        cout << "\n You chose a character.\n";
    }
    return 0;
}
```

→ Catch all exceptions:

- Sometimes, it is not possible to design individual exceptions for all possible cases.
- In such cases, a single "catch" block is sufficient.
- Syntax is as given below:

```
try
{
    ..................;
}
catch (...)        //accepts all types of acce exceptions.
{
    ..................;
}
```