Object-Oriented Programming using C++

C++ Notes Day-7 Date: 15-12-2024

Lets Revise

- Constant variable, data member, member function and object
- mutable data member.
- Reference
- Call by value vs call by address vs call by reference
- Difference between pointer and reference
- Exception handling in C++
- OOPS Concepts(major and minor pillars)
- Association, Composition and Aggregation
- Inheritance concept
    - Mode of inheritance
    - Types of inheritance
- Diamond Problem
- Virtual base class
- Upcasting
- Downcasting
- Virtual function
- Function overriding
- Early Binding and Late Binding

Inheritance and Related Concepts

- Object slicing
- Pure Virtual Function / Abstract class Demo
    - According to client's requirement, If implementation of base class member function is logically 100% complete then we should not declare base class member function virtual.
    - According to client's requirement, If implementation of base class member function is logically incomplete / partially complete then we should declare base class member function virtual.
    - According to client's requirement, If implementation of base class member function is logically 100% incomplete then we should declare base class member function pure virtual.
    - If we equate virtual function to 0 then such virtual function is called as pure virtual function.
    - Example:

```cpp
#include <iostream>
using namespace std;
class Demo1  //Abstract Class
{
public:
    int Num1;
    Demo1()
    {
        this->Num1=10;
    }
    virtual void Method1()=0;    //Pure Virtual Function
    virtual void Method2()
    {
        cout<<"Method-2 of Base"<<endl;
    }
    void Method3()
    {
        cout<<"Method-3 of Base"<<endl;
    }
};
class Demo2: public Demo1
{
public:
    int Num2;
    Demo2()
    {
        this->Num2=20;
    }
    void Method1()
    {
        cout<<"Method-1 of Derived"<<endl;
    }

};
int main()
{
    //Demo1 d2;  //NOT OK, as Demo1 has pure virtual function so it can't be
instantiated
    //Demo1 *ptr=new Demo();

    Demo2 d2;
    d2.Method1();

    Demo1 *ptr=new Demo2();

    ptr->Method2();
    return 0;
}
```

- We can not provide body to the pure virtual function. Hence it is also called as abstract method.
- If class contains at least one pure virtual function then such class is called as abstract class.
- If class contains all pure virtual function then such class is called as pure abstract class / interface.
- We can not create object of abstract class and interface but we can create pointer / reference of it.
- It is mandatory to override pure virtual function in derived class otherwise derived class can be considered as abstract.
- If we create object of derived class then constructor and destructor of base class gets called. Hence abstract class can contain constructor as well as destructor.
- Example:

```
#include <iostream>
using namespace std;
class Demo1   //Pure Abstract class
{
public:
    int Num1;
    Demo1()
    {
        this->Num1=10;
    }
    virtual void Method1()=0;    //Pure Virtual Function: Abstract Function
    virtual void Method2()=0;    //Pure Virtual Function: Abstract Function

};
class Demo2: public Demo1
{
public:
    int Num2;
    Demo2()
    {
        this->Num2=20;
    }
    void Method1()
    {
        cout<<"Method-1 of Derived"<<endl;
    }
    void Method2()
    {
        cout<<"Method-2 of Derived"<<endl;
    }
};
int main()
{
    //Demo1 d2;  //NOT OK, as Demo1 has pure virtual function so it can't be
instantiated
    //Demo1 *ptr=new Demo();

    Demo2 d2;
    d2.Method1();
    return 0;
}
```

- Example:

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

class Shape     //Abstract Class
{
public:
    double Area;
    virtual void CalculateArea()=0;
    void PrintArea()
    {
        cout<<"Area is:     "<<this->Area<<endl;
    }
};
class Rectangle: public Shape
{
public:
    double Length;
    double Breadth;
    Rectangle(double Length, double Breadth)
    {
        this->Breadth=Breadth;
        this->Length=Length;
    }
    void CalculateArea()
    {
        this->Area=this->Length*this->Breadth;
    }
};
class Circle: public Shape
{
public:
    double Radious;
    Circle(double Radious)
    {
        this->Radious=Radious;
    }
    void CalculateArea()
    {
        this->Area=3.14*this->Radious*this->Radious;
    }
};
int main()
{
    Rectangle r1(233.78, 345.89);
    Shape *ptr=&r1;
    ptr->CalculateArea();
    ptr->PrintArea();

    Circle c1(100.89);      //Statically Allocated Memory

    ptr=new Circle(234.90); //Dynamically Allocated Memory

    ptr->CalculateArea();
    ptr->PrintArea();

    ptr=&c1;

    delete ptr;

}
```

Templates and Introduction to STL in C++

- Template
    - In C++, if we want to write typesafe generic code then we should use template.
    - template is keyword in C++.
    - In C++, by passing data type as a argument, we can write generic code. Hence paramerized type is called as template.
    - Example:

```
template<typename T> //T: Type Parameter
void swap_object( T &object1, T &object2 ){
T temp = object1;
object1 = object2;
object2 = temp;
}
int main( void ){
char ch1 = 'A';
char ch2 = 'B';
swap_object<char>( ch1, ch2 );
//char: Type argument
//ch1,ch2: function argument
cout << "ch1 : " << ch1 << endl;
cout << "ch2 : " << ch2 << endl;
return 0;
}
```

- Example:

```
template<typename T>
T Add(T Num1, T Num2)
{
    return Num1+Num2;
}

int main()
{
    int Num1=100;
    int Num2=200;

    int Res=Add(Num1,Num2);
    cout<<"Sum of Two int is :  "<<Res<<endl;

    float Resf=Add(100.50f, 200.56f);
    cout<<"Sum of Two Float is :    "<<Resf<<endl;


    cout<<"Sum of Two Double is :   "<<Add(345.90,789.90)<<endl;
}
```

    - We can use typename and class keyword interchangably.

```
template<class T> //T: Type Parameter
void swap_object( T &object1, T &object2 ){
T temp = object1;
object1 = object2;
object2 = temp;
}
```

    - Process of identifing type and passing it as a argument implicilty to the function is called as type inference.

```
swap_object<char>( ch1, ch2 ); //OK
swap_object( ch1, ch2 ); //OK\
```

    - Template feature is designed for the data structure.
    - Using template, we can not reduce code size or execution time rather we can reduce developers effort/ development time.
- Types of template:
    - Function Template
    - Class Template
- Function template

```
template<class T> //T: Type Parameter
void swap_object( T &object1, T &object2 ){
 T temp = object1;
 object1 = object2;
 object2 = temp;
}
int main( void ){
 int a = 10;
 int b = 20;
 swap_object<int>( a, b );
 cout << "a : " << a << endl;
 cout << "b : " << b << endl;
 return 0;
}
```

- Class Template

```
#include <iostream>
using namespace std;
template<class T>
class MathOperations
```

```
{
public:
    T Val1;
    T Val2;
    void GetData()
    {
        cout<<"Enter 1st Element:   "<<endl;
        cin>>this->Val1;
        cout<<"Enter 2nd Element:   "<<endl;
        cin>>this->Val2;
    }
    void CompareData()
    {
        if(this->Val1>this->Val2)
        {
            cout<<"1st Element is Greater"<<endl;
        }
        else if(this->Val1<this->Val2)
        {
            cout<<"2nd Element is Greater"<<endl;
        }
        else
        {
            cout<<"1st and 2nd Element are equal"<<endl;
        }
    }
};
int main()
{
    MathOperations<int> in;
    in.GetData();
    in.CompareData();
    MathOperations<float> fl;
    fl.GetData();
    fl.CompareData();
    return 0;
}
```

Standard Template Library(STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions. STL has 4 components:
    - Algorithms
        - https://en.cppreference.com/w/cpp/algorithm
    - Containers
        - https://en.cppreference.com/w/cpp/container
    - Functors
        - https://en.cppreference.com/w/cpp/utility/functional
    - Iterators
        - https://en.cppreference.com/w/cpp/iterator
- C++ STL Containers

    - A container is an object that stores a collection of objects of a specific type.
    - Types of STL Container in C++
        - Sequential Containers
        - Associative Containers
        - Unordered Associative Containers
- Demo of Menu-Driven Student Management Application in C++ using

```
#include <iostream>
#include <vector>
using namespace std;
class Student
{
public:
    int Sid;
    string Name;
    Student()
    {

    }
    Student(int Sid, string Name)
    {
        this->Sid=Sid;
        this->Name=Name;
    }
    void ShowData()
    {
        cout<<"Id: "<<this->Sid<<" Name: "<<this->Name<<endl;
    }
};
class Admin
{
public:

    vector<Student> slist;   //Association

    Admin()
    {
        slist.push_back(Student(101,"Malkeet"));
        slist.push_back(Student(102,"Dipesh"));
        slist.push_back(Student(103,"Ritesh"));
    }
    void ViewStudents()
    {
        for(Student s:slist)
        {
            s.ShowData();
        }
    }
    void AddStudent()
    {
        Student s;
        cout<<"Enter Id:";
        cin>>s.Sid;
        cout<<"Enter Name:";
        cin>>s.Name;

        slist.push_back(s);
    }
    void DeleteStudent()
    {
        int id;
        cout<<"Enter Id:";
        cin>>id;
        for(int i=0;i<slist.size();i++)
        {
            if(slist[i].Sid==id)
            {
                slist.erase(slist.begin()+i);
                break;
            }
        }
    }
    void UpdateStudent()
    {
        int id;
        cout<<"Enter Id:";
        cin>>id;
        for(int i=0;i<slist.size();i++)
        {
            if(slist[i].Sid==id)
            {
                cout<<"Enter Name:";
                cin>>slist[i].Name;
                cout<<"Student Updated"<<endl;
                break;
            }
        }
    }
};
int main()
{
    Admin adm;
    adm.ViewStudents();
    adm.AddStudent();
    adm.ViewStudents();
    adm.DeleteStudent();
    adm.ViewStudents();
    adm.UpdateStudent();
    adm.ViewStudents();
}
```

To be discussed tomorrow (16-12-2024)

- Dynamic memory management using new and delete

- Difference between malloc and new
- Destructor
- Shallow Copy and Deep Copy
- Copy Constructor
- Friend Function
- Friend Class