**Name:** Umair Abbas

**Registration #:** L1F19BSCS0442

**Section:** D

Object Oriented Programming – Fall 2020

# CP Task 05 - Polymorphism

**NOTE:** This CP task is purely related to the READING HOMEWORK (Chapter 13) that is assigned to you earlier this week. If you have completed it already then you will be able to answer following questions otherwise first read the chapter then try to solve it.

**Question # 1:**

How does polymorphism promote extensibility?

**Answer:**

Polymorphism promotes extensibility by allowing new sub-classes and methods to be added to a class hierarchy without having to modify application programs that already use the hierarchy’s interface.

For example:  a Shape class contains a method called area ().

**Question # 2:**

Discuss the problems of programming with ***switch*** logic. Explain why polymorphism can be an effective alternative to using ***switch*** logic?

**Answer:**

One way to determine the type of the object is to use a switch case statement to check the value of a field in the object. This allows us to distinguish among object types. However, there are many potential problems which we have to face while using switch case like a programmer might forget to include a type test when one is warranted or might forget to test all possible cases. Every addition and deletion in switch case requires changings in switch case so here polymorphism comes in handy as while implementing polymorphism we don’t have to update classes again and again.

**Question # 3:**

1. What is **v-table**?
2. Distinguish between static binding and dynamic binding. Explain the use of **virtual functions** and the **v-table** in dynamic binding.

**Answer(i):**

v-table is an implementation detail used by the implementation to implement virtual functions/dynamic binding or dynamic dispatch.

**Answer(ii):**

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| Happens at compile time | Happens at run-time |
| Function definition and Function call are linked during compile time. | Functions calls are not bound until run-time. |
| Results in Faster Execution of a program. | It is flexible as it can handle different objects at run-time. |

The object of the class containing the virtual function contains a virtual pointer that points to the base address of the virtual table in memory. If there is a virtual function call, the v-table is used to resolve to the function address.

**Question # 4:**

While working with polymorphism, why we always need **virtual destructor** in base class? Code a scenario in which first show the problem when destructor in base class is not virtual then make it virtual and show that problem has resolved.

**Answer:**

Virtual destructor is necessary when different destructors have to follow proper order while objects is being deleted through base class pointer.

For Example: If your base class destructor is NOT virtual then only base class object will get deleted.

**Code without virtual destructor:**

class teacher {

  public:

    teacher()

    {

cout<<"Constructing teacher "<<endl;

}

    ~teacher()

    {

cout<<"Destructing teacher "<<endl;

}

};

class student: public teacher {

  public:

    student()

    {

cout<<"Constructing student "<<endl;

}

    ~student()

    {

cout<<"Destructing student "<<endl;

}

};

int main()

{

  student \*d = new student();

  teacher \*b = d;

  return 0;

}

**Output:**

Constructing teacher

Constructing student

Destructing teacher

**Code with virtual destructor:**

class teacher {

  public:

    teacher()

    {

cout<<"Constructing teacher "<<endl;

}

    Virtual ~teacher()

    {

cout<<"Destructing teacher "<<endl;

}

};

class student: public teacher {

  public:

    student()

    {

cout<<"Constructing student "<<endl;

}

    ~student()

    {

cout<<"Destructing student "<<endl;

}

};

int main()

{

  student \*d = new student();

  teacher \*b = d;

  return 0;

}

**Output:**

Constructing teacher

Constructing student

Destructing student

Destructing teacher

**Question # 5:**

What is the difference between **upcasting** and **downcasting**? Give a full code example for each.

**Answer:**

**Up casting:**

Up casting allows us to treat the child class object as if it were the parent class object. There are two ways to implement it. One is by creating parent class pointer and assigning it to the base classes reference and another by creating Parent classes referenced object and assigning it to the child class object.

**Code:**

class Parent

{

public:

void print()

{

cout << "Parent Class" << endl;

}

};

class Child : public Parent

{

public:

void print()

{

cout << "Child Class" << endl;

}

};

int main()

{

Parent \*p\_obj;

Child c\_obj;

p\_obj = &c\_obj;

p\_obj->print();

return 0;

}

**Output:**

Parent class

**Down Casting:**

It is vice versa of up casting. In this type of casting we create a parent class pointer reference to parent class reference.

**Code:**

class Parent

{

public:

void print()

{

cout << "Parent Class" << endl;

}

};

class Child : public Parent

{

public:

void print()

{

cout << "Child Class" << endl;

}

};

int main()

{

Parent \*p\_obj;

Child \*c\_obj = (Child\* ) &p\_obj;

c\_obj->print();

return 0;

}

**Output:**

Child Class

**Question # 6:** (Not from chapter 13)

1. Define type casting in C++?
2. What are 4 typecast operators in C++?
3. Distinguish between **type casting** and **type conversion**?

**Answer (i):**

When you assign a value of one primitive data type to another type is known as type casting.

**Answer (ii):**

Four typecast operators in C++ are as follow:

1. dynamic\_cast
2. reinterpret\_cast
3. static\_cast
4. const\_cast.

**Answer(iii):**

|  |  |
| --- | --- |
| **Type Casting** | **Type Conversion** |
| One data type is assigned to another by the user, using a cast operator then it is called Type Casting. | Conversion of one data type to another automatically by the compiler is call Type Conversion. |
| Type casting can also be applied to two incompatible data types. | Type conversion can only be implemented when two data types are 'compatible'. |
| a casting operator '()' is required. | No operator Required. |
| It is done During program designing. | It is done explicitly while compiling. |