Lecture 16.1

Topics

1. **Pure Virtual Function – Introduction**
2. **Abstract Classes – Introduction**
3. **Base Pointers, Derived Pointers and Objects – Examples**

____

1. Pure Virtual Function and Abstract Class

To ensure that a derived class must provide its VF definition, a **pure virtual function** is used.

1.1 Definition

A **pure virtual function** (**PVF**) has no definition relative to the base class. A **PVF** requires a prototype and tagged as follows,

```
virtual ReturnType functionName( argumentList ) = 0;
```

When the derived class does not override this **PVF**, an error will result. Thus, it forces the derived to provide the necessary definitions. This also implies that a class with at least one **PVF** is an incomplete class and will be used as base in creating derived classes.

A class with at least one **PVF** is called an abstract class. Abstract class (AC) must be inherited and must not be used as-is. That means no object from this abstract class is allowed. However, pointer from this abstract class is allowed and it is important in achieving the so-called run time polymorphism.

1.2 Run Time Binding

A point about VF is that once virtual it is always virtual. That means its definition can be altered in any derived classes. What happen when VFs are used?

- When a regular `function` (not VF) is declared and used, the compilation will provide information for calling. Thus, this early compile time (early binding) type of `function` is a fast call but is not flexible.
- While a late binding refers to events that occur at run time, a late bound `function` call is the one in which the address of the `function` to be called is not known until the program runs. VF in C++ is a late bound `function` event.
- When a VF is accessed through a base pointer (BP), the program must determine at run time what type of object it is pointing to and then it will select the appropriate `function`. This type of binding is flexible but slower because of the overhead tasks in tracing through the maze of overriding, addresses, etc.

2. Abstract Classes – Introduction

An abstract class is a class that has at least one pure virtual function as a member.

An abstract class is meant to be used as the base of some class hierarchies, which will have their virtual functions defined and given in each of the derived classes.

Let's consider a class **`Shape`**, which has two pure virtual member functions `getArea()` and `getPerimeter()`. The class **`Shape`** is the base for some other geometrical objects such as a point, a line, a triangle, a rectangle, etc.

Examples will be given to clarify the use of PVFs, abstract classes, and the class relationships.

3. Base Pointer, Derived Pointer, & Objects – Examples

Pointer is an important variable that is unavoidable in great many applications. Coupling with inheritance, the use of pointer provides flexibility and critical access to data.

Let's consider some examples of the base pointer (BP), derived class pointer, and objects. Recall that a BP is a pointer variable that was declared from a base class. This BP can be used to point to any base object as well as any other object created from one of the classes in the hierarchy.

That means with

```
class B;                // Base class

class D : public B;     // Derived class
```

then

```
B* bPtr;       // Base Pointer (BP)BP
B bObj;        // Assuming default constructor available
D dObj;        // Assuming proper constructors available


bPtr = &bObj; // Setting bPtr to point to a B object
bPtr = &dObj; // Setting bPtr to refer to a D object
```

Although a BP points to a derived object, it can only access to the inherited members from the base class with appropriate specification. **The BP always has and keeps the declared type of the base class**. It only understands and has access to the members from the base class. It does not have any notion about the **add-on** portion from the derived class.

Note that

1. A BP can be declared and initialized to point to a derived object,
2. It is not allowed to declare a pointer from a derived class (derived pointer, DP) and then to have it assigned and pointed to a base class object.

   The reason is quite simply that a DP can be used to access the members that come from the derived class (with proper access specification). Clearly, the base object with its existing memory structure will not provide any information for these add-on members. Thus, errors will be generated if one tries to do this!
3. Another point to remember about BP is that a pointer arithmetic applied to a BP will always be relative to the base object (or memory structure).

   Therefore, if a BP was pointing to a derived object and then got incremented, it will not point to the next derived object (as in an array of derived objects). Instead, it will be pointing to the memory block that, to the system, is the starting for the next base object. This is a sure trouble to be encountered for the next operation.

Let's consider an example below.

Example 1

```
//Program Name : cis25L1611.cpp
//Discussion :   Base Pointers

#include <iostream>
using namespace std;

class B {
```

```
public:
  B() {
    x = 0;
  }
  void setX(int a) {
    x = a;
  }
  int getX() {
    return x;
  }
private:
  int x;
};

class D : public B {
public:
  D() {
    y = 0;
  }
  void setY(int a) {
    y = a;
  }
  int getY() {
    return y;
  }
private:
  int y;
};

int main() {
  B bObj;
  B* bPtr;
  D dObj;

  bPtr = &bObj;
  bPtr->setX(5);
  cout << "\nBase object : bObj "
       << "\n\tx : " << bPtr->getX() << endl;

  bPtr = &dObj;
  bPtr->setX(15);
  dObj.setY(25);
  cout << "\nDerived object : dObj "
       << "\n\tx : " << bPtr->getX()
       << "\n\ty : " << dObj.getY() << endl;

  return 0;
}
```

OUTPUT

```
Base object : bObj
        x : 5

Derived object : dObj
        x : 15
        y : 25
```

Example 2

```
/**
 *Program Name: cis25L1612.cpp
 *Discussion:   Virtual Functions & Base Pointer
 */
#include <iostream>
#include <cmath>
using namespace std;

class Point {
public:
  Point() {
    dX = 0.0;
    dY = 0.0;
  };
  Point(double d1, double d2) {
    dX = d1;
    dY = d2;
  }
  ~Point(){ };

  void setX(double old) {
    dX = old;

    return;
  }
  double getX() {
    return dX;
  }
  void setY(double old) {
    dY = old;
     return;
  }
  double getY() {
    return dY;
  }

  virtual char* getType() {
    return "Point";
  }
  virtual double computeArea() {
    return 0.0;
  }
  virtual double computeVolume() {
    return 0.0;
  }

  Point operator+(Point &oldPt) {
    Point ptTemp;

    ptTemp.dX = dX + oldPt.dX;
    ptTemp.dY = dY + oldPt.dY;

    return ptTemp;
  }
private:
  double dX;
  double dY;
};

class Circle : public Point {
```

```
public:
  Circle() {
    dZ = 0.0;
  }
  Circle(double d1, double d2, double d3) : Point( d1, d2 ) {
    dZ = d3;
  }
  ~Circle(){ }

  void setZ(double old) {
    dZ = old;
  }
  double getZ() {
    return dZ;
  }
  char* getType() {
    return "Circle";
  }

  double computeArea() {
    return 3.1416 * dZ * dZ;
  }

  double computeVolume() {
    return 0.0;
  }
  Circle operator+(Circle &cirOld) {
    Circle cirTemp;

    cirTemp.setX(getX() + cirOld.getX());
    cirTemp.setY(getY()+ cirOld.getY());
    cirTemp.setZ(sqrt( ( (computeArea() +
                  cirOld.computeArea()) / 3.1416)));

    return cirTemp;
  }
private:
  double dZ;
};

class Cylinder : public Circle {
public:
  Cylinder() {
    dH = 0.0;
  }
  Cylinder(double d1, double d2, double d3, double d4) :
          Circle(d1, d2, d3) {
    dH = d4;
  }
  ~Cylinder(){ }
  void setH(double old) {
    dH = old;
  }

  double getH() {
    return dH;
  }
  char* getType() {
    return "Cylinder";
  }
  double computeVolume() {
    return 3.1416 * getZ() * getZ() * dH;
  }
private:
  double dH;
};

int main() {
  int i;
  Point* arrayPtr[3];

  Point pt1(1.5, 2.5);
  Circle cir1(3.5, 4.5, 2.0);
  Cylinder cyl1(0.0, -2.5, 2.0, 2.0);

  arrayPtr[0] = &pt1;
  arrayPtr[1] = &cir1;
  arrayPtr[2] = &cyl1;

  cout <<"\n" <<  arrayPtr[0]->getType() << " : "
       << "\n\tArea\t" << arrayPtr[0]->computeArea()
       << "\n\tVolume\t" << arrayPtr[0]->computeVolume()
       << endl;
  cout << "\n" << arrayPtr[1]->getType() << " : "
       << "\n\tArea\t" << arrayPtr[1]->computeArea()
       << "\n\tVolume\t" << arrayPtr[1]->computeVolume()
       << endl;

  for (i = 0; i < 3; i++) {
    cout << "\n" << arrayPtr[i]->getType() << " : "
         << "\n\tArea\t" << arrayPtr[i]->computeArea()
         << "\n\tVolume\t" << arrayPtr[i]->computeVolume()
         << endl;
  }

  cir1 = cir1 + cir1;

  cout << "\n" << arrayPtr[1]->getType() << " : "
       << "\n\tArea\t" << arrayPtr[1]->computeArea()
       << "\n\tVolume\t" << arrayPtr[1]->computeVolume()
       << endl;

  return 0;
}
```

OUTPUT

```
Point :
        Area    0
        Volume  0

Circle :
        Area    12.5664
        Volume  0

Point :
        Area    0
        Volume  0

Circle :
        Area    12.5664
        Volume  0

Cylinder :
        Area    12.5664
        Volume  25.1328

Circle :
        Area    25.1328
        Volume  0
```