Software Engineering 2 (C++)

CSY2006 (Week 17)

Dr. Suraj Ajit

Polymorphism and Virtual Member Functions

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

```
virtual void Y() {...}
```

- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Consider this function (from Program 15-10)

```
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Because the parameter in the `displayGrade` function is a GradedActivity reference variable, it can reference any object that is derived from GradedActivity. That means we can pass a GradedActivity object, a FinalExam object, a PassFailExam object, or any other object that is derived from GradedActivity.

A problem occurs in Program 15-10 and Graded Activity Version2 however...

Program 15-10

```
#include <iostream>
#include <iomanip>
#include "PassFailActivity.h"
using namespace std;

// Function prototype
void displayGrade(const GradedActivity &);

int main()
{
   // Create a PassFailActivity object. Minimum passing
   // score is 70.
   PassFailActivity test(70);

   // Set the score to 72.
   test.setScore(72);

   // Display the object's grade data. The letter grade
   // should be 'P'. What will be displayed?
   displayGrade(test);
   return 0;
}
```

```
23
24  //*************************************************************
25  // The displayGrade function displays a GradedActivity object's *
26  // numeric score and letter grade.                              *
27  //*************************************************************
28
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned ‘C’ instead of ‘P’. This is because the GradedActivity class’s `getLetterGrade` function was executed instead of the PassFailActivity class’s version of the function.

Static Binding

- Program 15-10 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the GradedActivityclass's version 2 of the function.

- We can remedy this by making the function *virtual*.

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.

- At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.
- See: VirtualExample.cpp

Virtual Functions

```
1  #include <stdio.h>
2  #include <iostream>
3  using namespace std;
4  class A{
5  public:
6      void print(){
7          cout << "Hello" << endl;
8      }
9  };
10
11 class B: public A{
12 public:
13     void print()
14     {
15         cout << "Goodbye" << endl;
16     }
17 };
18 void display(A &obj){
19     obj.print();
20 }
21 int main(){
22     A a;
23     B b;
24     display(a);// prints Hello
25     display(b);// prints Hello in C++ but prints Goodbye in Java (dynamic binding)
26     // To make C++ print Goodbye for above you need to declare print as virtual in base class.
27 }
```

Virtual Functions

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

Updated Version of GradedActivity

```
6  class GradedActivity
7  {
8  protected:
9     double score;    // To hold the numeric score
10 public:
11    // Default constructor
12    GradedActivity()
13       { score = 0.0; }
14
15    // Constructor
16    GradedActivity(double s)
17       { score = s; }
18
19    // Mutator function
20    void setScore(double s)
21       { score = s; }
22
23    // Accessor functions
24    double getScore() const
25       { return score; }
26
27    virtual char getLetterGrade() const;
28 };
```

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

Programs 15-11 and GradedActivity Version 3 rectify the problem by making function virtual

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms.

Programs 15-12 and GradedActivity Version 3 demonstrates polymorphism by passing
objects of the GradedActivity and PassFailExam classes to the displayGrade function.

Program 15-12

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11    // Create a GradedActivity object. The score is 88.
12    GradedActivity test1(88.0);
13
14    // Create a PassFailExam object. There are 100 questions,
15    // the student missed 25 of them, and the minimum passing
16    // score is 70.
17    PassFailExam test2(100, 25, 70.0);
18
19    // Display the grade data for both objects.
20    cout << "Test 1:\n";
21    displayGrade(test1);    // GradedActivity object
22    cout << "\nTest 2:\n";
```

```
23      displayGrade(test2);     // PassFailExam object
24      return 0;
25   }
26
27   //*********************************************************
28   // The displayGrade function displays a GradedActivity object's *
29   // numeric score and letter grade.                        *
30   //*********************************************************
31
32   void displayGrade(const GradedActivity &activity)
33   {
34      cout << setprecision(1) << fixed;
35      cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37      cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39   }
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a **reference variable or a pointer** (Pr 15-13), as demonstrated in the `displayGrade` function.

Polymorphism Requires References or Pointers

```
3  class Shape {
4     protected:
5        int width, height;
6     public:
7        Shape( int a = 0, int b = 0){
8           width = a;
9           height = b;
10       }
11       int area() {
12          cout << "Parent class area :" <<endl;
13          return 0;
14       }
15 };
16 class Rectangle: public Shape {
17    public:
18       Rectangle( int a = 0, int b = 0):Shape(a, b) { }
19
20       int area () {
21          cout << "Rectangle class area :" <<endl;
22          return (width * height);
23       }
24 };
25 class Triangle: public Shape {
26    public:
27       Triangle( int a = 0, int b = 0):Shape(a, b) { }
28
29       int area () {
30          cout << "Triangle class area :" <<endl;
31          return (width * height / 2);
```

Polymorphism Requires References or Pointers

```
5   // Main function for the program
6   int main() {
7       Shape *shape;
8       Rectangle rec(10,7);
9       Triangle  tri(10,5);
0       // store the address of Rectangle
1       shape = &rec;
2       // call rectangle area.
3       shape->area();
4       // store the address of Triangle
5       shape = &tri;
6       // call triangle area.
7       shape->area();
8       return 0;
9   }
```

Polymorphism Requires References or Pointers

```
56  class Shape {
57     protected:
58        int width, height;
59
60     public:
61        Shape( int a = 0, int b = 0) {
62           width = a;
63           height = b;
64        }
65        virtual int area() {
66           cout << "Parent class area :" <<endl;
67           return 0;
68        }
69  };
70
```

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);

cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Base Class Pointers

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5  // Function prototype
6  void displayGrade(const GradedActivity *);
7  int main(){
8     // Constant for the size of an array.
9     const int NUM_TESTS = 4;
10
11    // tests is an array of GradedActivity pointers.
12    // Each element of tests is initialized with the
13    // address of a dynamically allocated object.
14    GradedActivity *tests[NUM_TESTS] =
15       { new GradedActivity(88.0),
16         new PassFailExam(100, 25, 70.0),
17         new GradedActivity(67.0),
18         new PassFailExam(50, 12, 60.0)
19       };
20    // Display the grade data for each element in the array.
21    for (int count = 0; count < NUM_TESTS; count++)
22    {
23       cout << "Test #" << (count + 1) << ":\n";
24       displayGrade(tests[count]);
25       cout << endl;
26    }
27    system("PAUSE");
28    return 0;
29 }
```

Base Class Pointers

- Base class pointers and references only know about members of the base class
  - So, you can't use a base class pointer to call a derived class function

- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.
- So, a virtual function is overridden, and a non-virtual function is redefined.
- See Program 15-14 for an example

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.
- See Program 15-15 and Program 15-16 for examples

Virtual Destructors

```
#include <iostream>
using namespace std;
// Animal is a base class.
class Animal{
public:
    // Constructor
    Animal()
        { cout << "Animal constructor executing.\n"; }

    // Destructor
    ~Animal()
        { cout << "Animal destructor executing.\n"; }
};
// The Dog class is derived from Animal
class Dog : public Animal{
public:
    // Constructor
    Dog() : Animal()
        { cout << "Dog constructor executing.\n"; }
    // Destructor
    ~Dog()
        { cout << "Dog destructor executing.\n"; }
};
int main(){
    Animal *myAnimal = new Dog;
     delete myAnimal;
    return 0;
}
```

virtual ~Animal()

Abstract Base Classes and Pure Virtual Functions

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```

- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class

Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function
- See Student, CsStudent and Program 15-17 for examples