Chapter 15

Inheritance, Polymorphism, and Virtual Functions

What Is Inheritance?
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Example: Insects
- In addition to the common insect characteristics, the bumble bee has its own unique characteristics such as the ability to sting.
- In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

The "is a" Relationship
- Inheritance establishes an "is a" relationship between classes
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

Inheritance – Terminology and Notation
- **Base class** (or parent) – inherited from
- **Derived class** (or child) – inherits from the base class
- **Notation:**
  ```cpp
  class Student // base class
  {
  ... 
  }
  class UnderGrad : public Student // derived class
  {
  ... 
  };
  ```

Back to the ‘is a’ Relationship
- An object of a derived class ‘is a(n)’ object of the base class
- **Example:**
  - an UnderGrad is a Student
  - a Mammal is an Animal
- A derived object has all of the characteristics of the base class
What Does a Child Have?

An object of the derived class has:
- All members defined in child class
- All members declared in parent class

An object of the derived class can use:
- All public members defined in child class
- All public members defined in parent class

Protected Members and Class Access

- **protected member access specification**: like private, but accessible by objects of derived class
- **Class access specification**: determines how private, protected, and public members of base class are inherited by the derived class

Class Access Specifiers

1. **public**: object of derived class can be treated as object of base class (not vice-versa)
2. **protected**: more restrictive than public, but allows derived classes to know details of parents
3. **private**: prevents objects of derived class from being treated as objects of base class.

Inheritance vs. Access

<table>
<thead>
<tr>
<th>Base class members</th>
<th>How inherited base class members appear in derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>private: x</td>
<td>x is inaccessible</td>
</tr>
<tr>
<td>protected: y</td>
<td></td>
</tr>
<tr>
<td>public: z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base class members</th>
<th>How inherited base class members appear in derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>private: x</td>
<td>x is inaccessible</td>
</tr>
<tr>
<td>protected: y</td>
<td></td>
</tr>
<tr>
<td>public: z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base class members</th>
<th>How inherited base class members appear in derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>private: x</td>
<td>x is inaccessible</td>
</tr>
<tr>
<td>protected: y</td>
<td></td>
</tr>
<tr>
<td>public: z</td>
<td></td>
</tr>
</tbody>
</table>

More Inheritance vs. Access

When Test class inherits from Grade class using public class access, it looks like this:

```
class Grade
private members:
  char letter;
  float score;
  void calcGrade();
  public members:
    void setScore(float);
    float getScore();
    char getLetter();
```

When Test class inherits from Grade class using protected class access, it looks like this:

```
class Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;
  public members:
    Test(int, int);
```

```cpp
class Test : public Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;
  public members:
    Test(int, int);
    void setScore(float);
    float getScore();
    char getLetter();
```
More Inheritance vs. Access

```
class Grade
private members:
char letter;
float score;
void calcGrade();
public members:
void setScore(float);
float getScore();
char getLetter();

When Test class inherits from Grade class using private class access, it looks like this:
```
Passing Arguments to Base Class Constructor

- `Square::Square(int side):Rectangle(side,side)`
- `derived class constructor` and `base class constructor`
- `derived constructor parameter` and `base constructor parameters`

Redefining Base Class Functions

- `Not the same as overloading – with overloading, parameter lists must be different`
- `Objects of base class use base class version of function; objects of derived class use derived class version of function`

Base Class

```cpp
class GradeActivity {
private:
    char letter; // To hold the letter grade
    double score; // To hold the numeric score
    void determineGrade(); // Determine the letter grade
group;
    // Default constructor
    GradeActivity();
    // Accessor functions
    void setScore(double); // (score = x)
    double getScore(); // returns score;
    // Member function
    void getLetterGrade(); const
    { return letter; }
};

// getScore function
void setScore(double score) { // (score = x)
    letter = determineLetter(score); // Determine the letter grade
    // Accessor function
    double getScore() const { return score; }
};
```

Note: `setScore` function

Derived Class

```cpp
//GradeActivity.h
class GradeActivity {
private:
    char letter; // To hold the letter grade
    double score; // To hold the numeric score
    void determineGrade(); // Determine the letter grade
group;
    // Default constructor
    GradeActivity();
    // Accessor functions
    void setScore(double); // (score = x)
    double getScore(); // returns score;
    // Member function
    void getLetterGrade(); const
    { return letter; }
};
```

Redefined `setScore` function

From Program 15-6

```cpp
// create a GradeActivity object
GradeActivity a;
// GradeActivity main:
char letter;
// Get the student's name:
cout << "Enter the student's name: ";
cin >> letter;
// Get the course:
cout << "Enter the course: ";
cin >> course;
// Get the score:
cout << "Enter the score: ";
cin >> score;
// Get the active letter grade
cout << "Enter the current letter grade: ";
```

Program Output (example input shown in bold):

```cpp
Enter the student's name: John Doe
Enter the course: Math 101
Enter the score: 87.5
Enter the current letter grade: A
```

- The current score is 87.50
- The current grade is A
Problem with Redefining

- Consider this situation:
  - Class `BaseClass` defines functions `x()` and `y()`. `x()` calls `y()`.
  - Class `DerivedClass` inherits from `BaseClass` and redefines function `y()`.
  - An object `D` of class `DerivedClass` is created and function `x()` is called.
  - When `x()` is called, which `y()` is used, the one defined in `BaseClass` or the redefined one in `DerivedClass`?

Class Hierarchies

- A base class can be derived from another base class.

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

```
void displayGrade(const GradedActivity& activity)
{
    cout << activity.getScore() << endl;
    cout << "The activity's letter grade is " <<
    << activity.getLetterGrade() << endl;
}
```

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 15-9 however...
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 15-9 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the `GradedActivity` class's version 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.

**Updated Version of GradedActivity**
- To make a function virtual, place the `virtual` keyword before the return type in the base class's declaration:

```cpp
virtual char getLetterGrade() const;
```
- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.
- The function is now virtual.
- The function also becomes virtual in all derived classes automatically.
If we recompile our program with the updated versions of the classes, we will get the right output, shown here: (See Program 15-10 in the book.)

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

Program 15-11 demonstrates polymorphism by passing objects of the GradedActivity and PassFailExam classes to the displayGrade function

Polymorphism Requires References or Pointers

Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the displayGrade function

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;
```
Example

```cpp
void displayGrade( const GradedActivity *a );
...
GradedActivity *tests[NUM_TESTS] =
{ new GradedActivity( 88.0 );
  new PassFailExam( 100, 25, 70.0 );
  new GradedActivity( 67.0 );
  new PassFailExam( 50, 12, 60.0 );
};
...
for ( i = 0; i < NUM_TESTS; i++ )
displayGrade( tests[i] );
```

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.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler performs static binding on the destructor if the class ever is derived from.
- See Program 15-14 for an example.

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.

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:
  ```cpp
  virtual void Y() = 0;
  ```
- The `= 0` indicates a pure virtual function.
- There can be no function definition in the base class.

Multiple Inheritance

- A derived class can have more than one base class.
- Each base class can have its own access specification in derived class's definition:
  ```cpp
  class cube : public square, public rectSolid;
  ```
Multiple Inheritance

- Arguments can be passed to both base classes' constructors:
  - `cube::cube(int side) : square(side), rectSolid(side, side, side);`
- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance

- Problem: What if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator ::
  - Compiler errors occur if derived class uses base class function without one of these solutions