Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

June 28, 2018

Polymorphism: Part III

- In this lecture we continue our study of the concept of **polymorphism**.
- In this lecture we shall learn about the concept of an **abstract base class**.
- Abstract base classes are useful for writing polymorphic software libraries.

1 Polymorphism: a review

- First, a comment about inheritance.
  1. We know how to write derived classes, with the appropriate constructors, destructors and assignment operators.
  2. We also know how to override methods in the base class.
  3. We know about the inheritance tree.
- Next, we turn to polymorphism.
  1. We now know how to declare classes with virtual methods.
  2. If a class has virtual methods, we also know the destructor should be tagged as virtual.
  3. The virtual destructor ensures that the memory is deallocated correctly in polymorphic dynamic memory allocation and release.
- With all of the above, we can write polymorphic code.
- For example, we can write a polymorphic software library, using only references and pointers to a collection of base classes.
  1. Users of the library derive their own classes from the base classes in our library, and override the virtual functions according to their needs.
  2. The library functions correctly, even though the library has no knowledge what the derived classes are.
  3. The derived classes do not yet exist, at the time the library is written.

2 Abstract base class

- There is one further development of polymorphism we can make.
- It is not technically *necessary*, but it is *useful*.
- This is the concept of an **abstract base class.**
- An **abstract base class** is a class such that *objects of the class cannot be instantiated.*
  1. All objects must be of derived classes only.
  2. We can only declare pointers and references to an abstract base class.
  3. Pointers and references are not objects, hence they are permitted to exist.
- *Why would anyone want to write a class such that objects of it cannot be instantiated?*
- As stated above, it is not *necessary*, but in some contexts it can be *useful*.
- **Abstract base classes are widely used in commercial software.**
- **I have written and used abstract base classes many times in my career.**

3 Abstract class

- In general, an **abstract class** is one which cannot be instantiated directly.
- If an abstract class is the base of an inheritance tree, then we call it an abstract "base" class.
- In practice, most examples of abstract classes are abstract base classes.

4 Abstract base class: interface

- An abstract base class is really an interface.
- Suppose we write a software library for an accounting system, to keep track of customers, suppliers, sales and receipts, income and expenses, inventory, etc.
- Our library might be of interest to a wide variety of organizations, for example department and retail stores, restaurants, freight/trucking companies, possibly a financial accounting system for stocks, or medical billing companies (although there may be strict legal regulations for such organizations, which do not apply to other types of businesses), etc.
- Overall, what our library does is provide a *service*, but we have no specific model for the objects our library acts on.
- Nevertheless, our library must operate on *something.*
- We cannot write virtual functions without a base class of some sort.
- In such a situation, it is helpful to employ an abstract base class.
    1. The abstract base class supplies a description of functionality (which is coded in the library functions).
    2. It is the users of the library who write derived classes and instantiate actual objects.
    3. We have no need to instantiate any objects ourselves.
- This is quite a common scenario in a polymorphic software library.
    1. If the entire library is written using pointers and references to a collection of base classes, there may be no need to instantiate any objects of the base classes themselves.
    2. The users will write derived classes and supply all the objects for the library to act on.

5 Abstract base class: comment

- How do we write an abstract base class?
- There is more than one way to do it.
- Furthermore, the way I prefer is not the standard technique described in the textbooks.
- Although my preferred style does work and is employed by experts.
- I read about it on a web page, and I liked it, for reasons I shall explain below.
- However, we begin with the standard textbook procedure, that of **pure virtual functions.**

6 Pure virtual function

- Let us declare a class `PVFname` ("PVF" for "pure virtual function") as follows.

  1. The class has a virtual destructor.
  2. The class has a virtual method "`x()`" which is also `const`.
  3. The class has a non-virtual method "`print()`" which internally calls virtual functions.
  4. Finally, the class has a **pure virtual function** "`name()`" which will be explained below.

```
class PVFname {
public:
  virtual ~PVFname() {}                  // virtual destructor

  virtual string name() = 0;             // pure virtual function

  virtual double x() const { return 0; }

  void print() {
    cout << x() << endl;                 // calls virtual function
    cout << name() << endl;              // calls pure virtual function
  }
};
```

- The notation "**= 0**" means that the method "`string`" has no function body.
- That is to say, a function body for the method "`string`" does not exist anywhere in the compiled project.
- **Therefore if a calling application attempts to instantiate an object of the class `PVFname`, the compiler will generate an error.**
- We denote a method with no function body by the term **pure virtual function.**

  1. A derived class must override the method `name()` and supply a function body for it.
  2. Then objects of the derived class can be instantiated.
  3. If a derived class inherits directly from `PVFname` and does not override the method `name()`, the derived class is also an abstract class.

- **Caveat:**

  1. C++ actually supports a mechanism whereby it is possible to write a "default implementation" for a pure virtual function.
  2. However, this takes us into complicated territory, beyond the scope of this course.
  3. A "default implementation" for a pure virtual function is advanced programming even for experts.

7 Classes Aname, Bname, Cname

- We declare an inheritance tree below, with `PVFname` as the base.
  1. `Aname` derives from `PVFname` and overrides the pure virtual function.
  2. `Bname` derives from `PVFname` but **does not override the pure virtual function.**
  3. `Bname` is an abstract class and objects of type `Bname` cannot be instantiated.
  4. `Cname` derives from `Bname` and overrides the pure virtual function.
  5. Therefore `Aname` and `Cname` are not abstract classes.

```
class Aname : public PVFname {
public:
  double x() const { return 1.1; }
  string name() { return "Aname"; }     // override pure virtual function
};

class Bname : public PVFname {          // abstract class
public:                                 // DOES NOT OVERRIDE pure virtual method
  double x() const { return 2.2; }
};

class Cname : public Bname {
public:                                 // not abstract
  string name() { return "Cname"; }     // override pure virtual function
};
```

- Here is a working main program and function to use the above classes.
- Just for show, a base class pointer is used instead of a reference.

```
#include <iostream>
#include <string>
using namespace std;
// class declarations

void pvf_func(PVFname *p)    // pointer to abstract base class
{ p->print(); }

int main()
{
  Aname a;
  Cname c;
  pvf_func(&a);              // polymorphic function call
  pvf_func(&c);              // polymorphic function call
  return 0;
}
```

8 Pure virtual functions: a personal critique

- I personally do not like pure virtual functions.
- I do not employ pure virtual functions in my own work.
- If an abstract base class has pure virtual functions, a derived class is obliged to override all the pure virtual methods, even if it has no meaningful functionality to offer for an override.
- In a commercial software package, this can be bothersome to customers, who are forced to write overrides for things they are not interested in. That can lead to customer dissatisfaction and loss of business. It may sound like a small detail, but it does matter.

9 *Is an abstract base class an incomplete class?*

- Some authors state that because an abstract class has one or more pure virtual functions, therefore an abstract class is an incomplete class definition.
- It is not only necessary to write a derived class, it is furthermore necessary for the derived class to override all the pure virtual functions (or to derive from a class in the inheritance tree which already overrides all the pure virtual functions).
- *What to say in response?*
    1. Let us declare an abstract base class with no pure virtual functions.
    2. That is my preferred way to declare an abstract base class.
    3. Let us furthermore declare a derived class which does not override any methods in the base class.

10 Abstract base class: protected constructors and assignment operator

- My preferred technique to declare an abstract base class is to **make the constructors and assignment operator protected.**
- Then external applications cannot instantiate objects of the class.
- The constructors and assignment operator must be protected (not private), so that derived classes can access them.
- The destructor can be public. (Remember it must be a virtual destructor.)
- Here is the declaration for `ABC` ("abstract base class").
  1. The class `ABC` cannot be instantiated directly.
  2. It does not have any pure virtual functions.
  3. Hence a derived class can inherit from it without being forced to override anything.

```
class ABC {
public:
  virtual ~ABC() {}               // virtual destructor

  virtual string name()    { return "unset"; }
  virtual double x() const { return 0; }

  void print() {
    cout << x() << endl;
    cout << name() << endl;
  }

protected:
  ABC() {}                        // protected constructors and assignment
  ABC(const ABC &orig) {}
  ABC operator= (const ABC &rhs) { return *this; }
};
```

- We declare three derived classes as follows.
  1. `Aabc` derives from `ABC` and **does not override anything.**
  2. `Babc` derives from `Aabc` and overrides `x()`.
  3. `Cabc` derives from `Babc` and overrides `name()`.
  4. Hence overall `Cabc` invokes overrides for both `x()` and `name()`.

- **See next page(s).**

11 Classes Aabc, Babc, Cabc

- The declarations for the derived classes `Aabc`, `Babc` and `Cabc` are given below.

```
class Aabc : public ABC {               // no overrides
public:
};

class Babc : public Aabc {
public:
  double x() const { return 1.1; }
};

class Cabc : public Babc {
public:
  string name() { return "Cabc"; }
};
```

- Here is a working main program and function to use the above classes.

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

// class declarations

void abc_func(ABC *p)        // pointer to abstract base class
{ p->print(); }

int main()
{
  Aabc a;
  Babc b;
  Cabc c;
  abc_func(&a);              // polymorphic function call
  abc_func(&b);              // polymorphic function call
  abc_func(&c);              // polymorphic function call
  return 0;
}
```