(12-1) OOP: Polymorphism in C++ D & D Chapter 12

Instructor - Andrew S. O'Fallon
CptS 122 (April 3, 2019)
Washington State University

Key Concepts

- Polymorphism
- `virtual` functions
- Virtual function tables

What is Polymorphism? (I)

- *Polymorphism* is the ability to use the same expression to denote different operations
- *Runtime polymorphism* is the ability to associate multiple meanings to a single function name though the use of late or dynamic binding
  - You can process objects of the same class hierarchy as if they are all objects of the hierarchy's *base* class
- *Compile time polymorphism* is the type that is achieved through function overloading, operator overloading, and templates
- Enables you to "program in the general", instead of "program in the specific"
- Another form is *parametric* polymorphism
  - the (data) type is left unspecified and later instantiated
  - templates provide parametric polymorphism

What is Polymorphism? (II)

- Provides a mechanism to allow programs to process objects of classes that are part of the same class inheritance hierarchy as though they are part of the base class
  - This way we can create several base-class pointers or references at compile and decide the specific object to which they point or reference at runtime
- Allows you to design and implement systems that are *extensible* – classes can be added with little to no modifications to portions of the program
- `virtual` functions provide a means to apply runtime polymorphism

Virtual Functions

- A *virtual* function is specified by using the keyword `virtual`
- A function whose behavior can be *overridden* or replaced
  - Function *overriding* is a feature that allows a derived class to provide a specific implementation for a function that is provided by a base class – this is NOT the same as function *overloading* – the return type, name, and parameters are the same in the base and derived classes

Pure Virtual Functions

- A *pure virtual* function is specified by setting the function "= 0" in the declaration
- Does not provide an implementation for the function, just a declaration
- Each derived class must *override* all base-class pure `virtual` functions with concrete implementations – this is not the case for virtual functions that are not pure
- The compiler will report an error if a pure virtual function is not overridden

Virtual Destructors

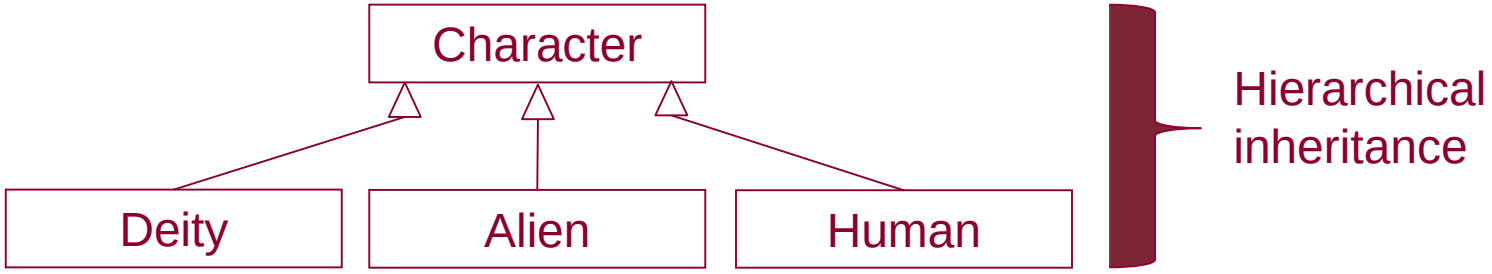
- Required if you need to `delete` an instance of a derived class through a base class pointer
- If the base class destructor is not `virtual`, then trying to delete the derived class object through a base class pointer may result in *undefined* behavior because only the base class destructor will be invoked

Abstract Classes

- A class is considered *abstract* if one or more of its `virtual` functions is *pure*
- Cannot be instantiated
- If you have decided that a class must be abstract, then you should make each function that must be overridden pure `virtual`
  - Remember: a “non-pure” virtual function does not have to be overridden!
- Remember classes that can be instantiated are called *concrete* classes

Hierarchical Inheritance - Inheritance Structure of Video Game Characters (I)

- Deity, Alien, and Human classes are derived from a base class Character:



Hierarchical Inheritance - Inheritance Structure of Video Game Characters (II)

- What should be in the base class Character?

```
class Character
{
        public:
                // Will not show setters, getters, constructors explicitly
                virtual ~Character (); // virtual destructor
                virtual void move (int x, int y);
                virtual void render ();
        private:
                int mPosX;
                int mPosY;
                Image mSprite;
};
```


Hierarchical Inheritance - Inheritance Structure of Video Game Characters (III)

- Should we define the Character class as an abstract class, i.e. a class that cannot be instantiated?
  - Will we ever instantiate a Character object? Or will we just instantiate Deity, Alien, and Human objects?
    - In this example, we will make our Character class abstract – we will use it as a general way to describe all characters in the game, but will not instantiate a Character object

```
class Character
{
        public:
                virtual void render () = 0; // pure virtual
        private:
};
```


Hierarchical Inheritance - Inheritance Structure of Video Game Characters (IV)

- Each derived class (Deity, Alien, and Human) will respond to function render () in a unique way
  - The same *message* (i.e. render ()) sent to different objects will provide many different results or forms – i.e. polymorphism
  - Making the function render () pure `virtual` ensures that each derived class provides its own implementation for it

Hierarchical Inheritance - Inheritance Structure of Video Game Characters (V)

- How does each of the derived classes declare a render () function?
  - These functions should have the same return type, name, and parameter list as the base class one; however, they don't need to be virtual unless we plan on overriding the functions in the derived classes as well (Zeus could be derived from Deity)

```
class Deity : public Character // public inheritance
{
        public:
                void render ();  // Does NOT necessarily need to be virtual
        private:
};
class Alien : public Character
{
        public:
                void render ();
        private:
};
class Human : public Character
{
        public:
                void render ();
        private:
};
```


Hierarchical Inheritance - Inheritance Structure of Video Game Characters (VI)

- What is the impact of `virtual` functions?
  - Well…let's look at the following code snippet:

```
Character *pGameChar = NULL;
…
pGameChar = new Alien;
…
pGameChar -> render (); // render () is virtual in the base class!
```

    - If render () was not declared as virtual in the Character base class, then a decision about which render () to invoke would be based on the pointer's or handler's type (i.e. Character *) – would not render an Alien!

Hierarchical Inheritance - Inheritance Structure of Video Game Characters (VII)

- What is the importance of a `virtual` destructor for this example?
  - Well…let's look at the following code snippet:

```
Character *pGameChar = NULL;
…
pGameChar = new Alien;
…
delete pGameChar;
```

    - The concern is that if the base class destructor (i.e. ~Character ()) is not virtual, then it's the one that is used to delete an Alien – this is problematic because an Alien has attributes (data members) that a general character does not – undefined behavior could result (memory leaks as well)!

Virtual Function Tables (I)

- Polymorphism introduces overhead
  - i.e. more memory consumption and processor time
- The compiler will build a `virtual` function table (vtable) for each class that has at least one virtual function – each instance of an object of the same class, uses the same table
  - An executing program uses the vtable for determining the proper implementation each time a virtual function is called
  - The determination of which function to call at *runtime* denotes *dynamic* binding

Virtual Function Tables (II)

- The vtable consists of pointers to each `virtual` function in a class
  - If the function is pure `virtual`, then the function pointer is set to 0 or NULL – indicates abstract class!

Virtual Function Tables (III)

- Three levels of *indirection* required to implement polymorphism
  - First level – the pointers to functions stored in the vtable
  - Second level - when an object of a class with one or more `virtual` functions is instantiated, the compiler inserts in the object a pointer to the associated vtable
  - Third level – pointers to the objects that are declared

Summary

- Polymorphism allows for the developer to “program in the general”

References

- P.J. Deitel & H.M. Deitel, *C++ How to Program (9th Ed.)*, Pearson Education, Inc., 2014.
- D.S. Malik, *C++ Programming: Program Design Including Data Structures (8th Ed.)*, Cengage Learning, 2018.
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013