ECE 3574: Dynamic Polymorphism using Inheritance

Changwoo Min

Administrivia

- Survey on class will be out tonight or tomorrow night
  - Please, let me share your idea to improve the class!

Meeting 10: Dynamic Polymorphism using Inheritance

- Today we will learn more about code reuse via dynamic polymorphism.
  - Subtyping and Object Hierarchies
  - Abstract Base Classes as Interfaces
  - Dynamic Casting
  - Qt Object Model
  - Qt Ownership
  - Examples
- There is no exercise today.

C++ Inheritance and Base Classes

- C++ has several mechanisms to reuse code.
- One of them is polymorphism (many-form), where a class can inherit methods from one or more other classes.
- Polymorphism means selecting what code to run based on the type.

Classic Shape Example

- Suppose we wanted to have classes that model closed 2D shapes. There are things that every 2D shape has, for example a perimeter. We can ensure that any class that implements a specific 2D shape has an appropriate method by first defining a base class

```
class Shape2DBase
{
public:
    virtual double perimeter() = 0;
};
```

Classic Shape Example

```
class Shape2DBase
{
public:
    virtual double perimeter() = 0; // abstract virtual function
};

void main(void) {
    Shape2DBase shape; // compiler error!
}
```

- `virtual`: *virtual function*
  - it can be redefined in subclasses
- `= 0`: *abstract* virtual function (similar to interface in Java)
  - this class does not provide an implementation on purpose
  - we can't instantiate such a class

Classic Shape Example

- We can define and implement a setup of classes that conform to the base class using public inheritance (there are other kinds we are ignoring for now). For example we might define a Circle as

```
class Circle: public Shape2DBase // public inheritance of Shape2DBase
{
public:
    Circle(double r): radius(r) {};
    double perimeter() { // overriding of the abstract virtual function
        return 2*M_PI*radius;
    }
private:
    const double radius;
};
```

- We might continue with classes for Square, Rectangle, Ellipse, etc.

Classic Shape Example

- This is handy because, while I can't instantiate the Shape2DBase, I can a pointer or a reference to one. So I could define a function that works for any subclass of Shape2DBase (lets say I want to show the perimeter).
- I can then pass a Circle, Square, etc to the function. Since it knows the classes have a perimeter method it can call it. Example

```
void show_perim(Shape2DBase & shape) {
    std::cout << "Perimeter = " << shape.perimeter() << std::
}

void main() {
    Circle c1(1.0);
    show_perim(c1);
}
```

Templates versus Base Classes

- You might have noticed this looks similar to templates. For example I could define Circle, Square, etc without inheritance but till defining a perimeter method, then define the function as a template.
- The difference is one between runtime and compile time, or dynamic versus static polymorphism.

```
template<typename T>
void show_perim(T & shape) {
    std::cout << "Perimeter = " << shape.perimeter() << std::
}
```

Dynamic versus static polymorphism

- Static polymorphism (also called compile-time polymorphism):
  - Pro: can check assumptions at compile time
  - Pro: allows the compiler to optimize
  - Con: leads to larger binary code, cannot switch code at runtime
- Dynamic polymorphism (also called Subtyping):
  - Pro: allows switching at runtime (for example based on input)
  - Pro: has smaller binary size
  - Con: requires runtime/dynamic casting
  - Con: can’t selectively optimize sine it happens after compilation

Polymorphism means selecting what code to run based on the type

- Dynamic polymorphism means the selection happens at runtime. The basic language mechanisms are:
  - Inheritance or Derivation
  - Virtual Functions, also called dynamic dispatch or runtime dispatch (virtual function table per class)
  - Encapsulation via private/public

Inheritance allows us to build one class from another

- Represents an "is-a" relationship. A `Circle` *is a* `Shape`. You specify the *base class* after declaring the *derived class* as

```
class BaseClass {};
class DerivedClass: public BaseClass {};
```

Inheritance allows us to build one class from another

- The public part means that the public members of `BaseClass` are inherited and become public members of `DerivedClass`. Note the above is equivalent to since struct members are public by default.
- Example: `inherit.cpp`

```
class BaseClass {};
class DerivedClass: BaseClass {};
```

Review: private, protected, and public

- members that are `private` can only be accessed by members in the same class
- members that are `protected` can only be accessed by members in the same class and those derived from it
- members that are `public` can be accessed by any functions
  - `struct`: a class that all members are `public`
- Note, this simplified version ignores *friend* access

private, protected, and public inheritance

- *private inheritance* means that the public and protected members of the base class can only be used by the derived class
- *protected inheritance* means that the public and protected members of the base class can only be used by the derived class and any classes derived from it
- *public inheritance* means that the public members of the base class can be used by any function
- *Note, there is no way to inherit the private members of a class.*

Summary: private inheritance

```
class BaseClass {};
class DerivedClass: private BaseClass {};
```

- It means that the the public and protected members of `BaseClass` become private members of `DerivedClass`.
- This is less common than the others, but can be useful for writing adapters.

Summary: protected inheritance

```
class BaseClass {};
class DerivedClass: protected BaseClass {};
```

- It means that the the public and protected members of `BaseClass` become protected members of `DerivedClass`.
- This is used for keeping functionality within the tree of objects.

Summary: public inheritance

```
class BaseClass {};
class DerivedClass: public BaseClass {};
```

- It means that the the public members of `BaseClass` become public members of `DerivedClass`.
- This is the most used, and presents the client code with a polymorphic interface.

A graphical view

Inheritance allow you to specify a tree relationship among types

Virtual Functions

- **Declaring a method virtual means that it can be overridden in the `DerivedClass`, but need not be.**
- Note that you can redefine the method in `DerivedClass` even if it is not marked virtual, but it will not be called when represented as the `BaseClass`.
- This is a source of much pain. If you intend for a method to be overridden, mark it as `virtual`.
- Note the virtual keyword is not used in the implementation if outside the class definition.

Abstract Virtual Functions

- To force the derived class to implement the method you make it pure as

```
struct BaseClass {
    virtual void aMethod() = 0;
};
```

- Abstract virtual function = pure virtual function
  - Similar to interface in Java
- Take care when overriding that the name and arguments match.

C++ allows multiple inheritance

```
class Base1 {};
class Base2 {};
class Derived: ACCESS Base1, ACCESS Base2
```

- where ACCESS can be private, protected, or public.
- This feature is much maligned because it creates confusion around what gets inherited.
- It is useful though for defining an interface.
  - Multiple inheritance other than interface is a code smell
  - See Qt class hierarchy
- Example: `multiple.cpp`

An interface is a base class with only pure virtual methods

- This allows you to express that a class implements that interface explicitly, and the compiler verifies that it at least implements the methods.
- When used with multiple inheritance this allows you to express that some parts of an object tree implement a certain interface.
- Example: `ADT_interface.h`

Heterogeneous Collections using a Base Type

- Since a pointer or reference to a base object can actually refer to a derived object, you can defined containers of mixed type (within the object hierarchy)
- Example: `collections.cpp`

Dynamic Casting

- Given a pointer to a base class you can attempt to convert it back to a derived type using `dynamic_cast`.
- This works for up-cat (but is unnecessary), down-cast, and sideways-cast.
- Example: `casting.cpp`
- Type conversions in C++
  - `dynamic_cast<>`, `static_cast<>`, `reinterpret_cast<>`, `const_cast<>`, and `typeid()`

An important point: any base class should define its destructor as virtual

- If you don't then if you call delete on a base pointer to a derived object, the derived destructor is not called.
  - any class designed to be used as a base class will have a virtual destructor
  - I repeat, do not derive from a class that has a non-virtual destructor.
- Example: `virtual.cpp`
- ECPP: Item 7: Declare destructors virtual in polymorphic base classes

Some remarks about dynamic polymorphism

- There are many places for bugs to hide
- Don’t get carried away with defining object hierarchies.
- Excessive casting is a code smell

Uses of Dynamic Polymorphism

- Dynamic Polymorphism is very useful when you want to treat objects (instances of classes) as hierarchical data.
  - Graphical user interfaces are ideally suited to polymorphism. They are just trees of widgets. Code to layout and draw widgets should be independent of the specific interface.
  - Rendering of dynamic objects, for example in simulations or games, should be independent of the specific objects involved.
  - Employees are people that have categories and form departments, units, divisions, etc.

Qt Object Hierarchy

- In Qt `QObject` is the base of everything. This is critical to how the properties and signals/slots are implemented.
- So, any class you write that needs to communicate with Qt should derive from `QObject`.
- Simple example: a class that can receive a print signal from another `QObject`.

```
class Printer: public QObject
{
Q_OBJECT
public slots:
    void print();
};
```

Qt Memory Ownership

- Qt objects form trees so that each object has a parent.
- Heap allocated objects are owned by their parents and destroyed when their parents are.
- For stack allocated objects take care the parent object is instantiated before children.

Next Actions and Reminders

- Read about Composition