Lecture 3

BASIC POLYMORPHISM

Today's class:

- Inheritance and Abstract Base Classes
- Introduce HW 2 and ICE 2.

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.

This has several uses, but the one that concerns us at the moment is specifying an *interface*, a class where the public methods are defined but not implemented.

- This defines the way client code can use a class that conforms to the interface.
- To define such a class you inherit from the interface, called a *base class* in C++, and implement the methods.

Shape Example

Suppose we wanted to have classes that model closed 2D shapes.

- There are things that (almost) 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;
};
```

- Note the use of the keyword `virtual` which means it can be redefined in subclasses and the = 0 syntax which says this class does not provide an implementation on purpose (Abstract Base Class). Defined this way we can't instantiate such a class – the following will not compile:

```
Shape2DBase shape;
```

Classic Shape Example

- We can define and implement a set of classes that conform to the base class using *public inheritance*
- For example we might define a Circle as

```
class Circle: public Shape2DBase
{
public:

  Circle(double r): {radius = r};

  double perimeter()
  {
    return 2*M_PI*radius;
  }

private:
  double radius;
};
```

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

Classic Shape Example

- This is handy
  - I can create pointer of a Shape2DBase.
  - I can define a function that works for any subclass of Shape2DBase
    - For example – printing the perimeter:

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

  - I can then pass a Circle, Square, etc to the function (Dynamic Binding).
    - It knows the classes have a perimeter method it can call.

```
Circle c1(1.0);

show_perim(c1);
```

Templates versus Base Classes

- Abstract classes may look similar to templates.
- Define Circle, Square, etc. without inheritance
  - Define the perimeter function as a template

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

- The difference is between runtime (abstract classes) and compile time (template) resolution
- Dynamic versus static polymorphism.
- Use inheritance for “is a” relationships
- Use templates for “works with” relationships

In Class Exercise (due Tuesday)

- Now, supplied with templates and the notion of base classes we can create an interface for the generic Bag ADT and adapt our implementation of Bag to use this interface definition.
    1. Download the starter code
    2. In the file abstract_bag.hpp define a C++ interface for our Bag ADT.
    3. Adapt the Bag implementation using automatic storage in the files bag_simple.hpp and bag_simple.tpp to use this interface
    4. Build your code locally as you work.
    5. Submit your code (critically abstract_bag.hpp and modified bag_simple.hpp) files via Canvas.

Assignment/Homework

- Reading : Carrano pp. 95-114, 117-132
- HW1 & ICE1 due **Today.**
- P1 due Thursday.
- HW2 released, due on Tuesday