ECE 2574: Data Structures and Algorithms - Basic Polymorphism

C. L. Wyatt

Today we will look at templates (again) and inheritance, two powerful mechanisms for code reuse in C++.

- Warmup
- More about Templates
- C++ Inheritance and Base Classes
- Exercise: Defining a Bag Interface in C++

Generics in C++

- Templates elevate types to be generic, named but unspecifed, and can work with functions and classes.
- Templates allow code reuse as long as the types meet the functionality required by the template
- The C++ standard library uses templates extensively

Example 1: template function to swap

A simple example is a function to swap the contents of two variables (similar to std::swap):

```
template< typename T >
void swap(T& a, T & b)
{
  T temp(b);
  b = a;
  a = temp;
}
```

Example 1: template function to swap

The symbol `T` acts like a variable, in fact it is a type variable. Defined this way swap is generic, I can use it on any type that can be copied. For example:

```
int a = 1;
int b = 2;

std::cout << a << ", " << b << std::endl;
swap(a,b);
std::cout << a << ", " << b << std::endl;

std::string A = "foo";
std::string B = "bar";

std::cout << A << ", " << B << std::endl;
swap(A,B);
std::cout << A << ", " << B << std::endl;
```

Example 1: template function to swap

If the type does not support a particular usage it generates a compile time error. For example suppose I wrote a class that explicitly does not allow copies

```
class NoCopy
{
public:
  NoCopy() = default;
  NoCopy(const NoCopy & x) = delete;
};
```

and tried to use swap as

```
NoCopy x,y;
swap(x,y);
```

My compiler complains

```
swapexample.cpp:7:5: error: call to deleted constructor of
  T temp(b);
```

Example 2: template class to hold a pair of objects

Templates work with classes as well. For example, we might define a tuple holding two different types (aka std::pair) as

```
template <typename T1, typename T2>
class pair
{
public:

  pair(const T1 & first, const T2 & second);

  T1 first();
  T2 second();

private:
  const T1 m_first;
  const T2 m_second;
};
```

Example 2: template class to hold a pair of objects

And implement it like

```
template <typename T1, typename T2>
pair<T1,T2>::pair(const T1 & first, const T2 & second)
: m_first(first), m_second(second)
{}

template <typename T1, typename T2>
T1 pair<T1,T2>::first()
{
  return m_first;
}

template <typename T1, typename T2>
T2 pair<T1,T2>::second()
{
  return m_second;
}
```

Example 2: template class to hold a pair of objects

We might use it like so

```
pair<int,std::string> x(0, std::string("hi"));

std::cout << "First = " << x.first() << std::endl;
std::cout << "Second = " << x.second() << std::endl;
```

Warmup #1

The C++ standard library includes several classes called *containers*. Look up the definition for `std::vector`, one such container. Which of the following are correct ways to declare a variable named myvec with a type representing a vector of vectors of integers?

- `std::vector< std::vector<int> > myvec;` (86%) CORRECT
- `std::vector<std::vector<int> > myvec;` (79%) CORRECT
- `vector<vector<int> > myvec;` (21%)
- `std::vector< std::vector<int>> myvec;` (54%)

Warmup #1

```
std::vector< std::vector<int> > myvec;
```

- note, vector is in the namespace std
- take care to include a space between ‘$> >$’ in nested templates as the compiler gets confused with the stream extraction operator ‘$>>$’.

Warmup #2

Why does your text suggest including `PlainBox.cpp` at the bottom of `PlainBox.h`?

- You can always include an implementation file in a header (2%)
- To make the code compile faster (5%)
- To make the code run faster (4%)
- Because it implements the template member functions (89%) CORRECT

Warmup #3

Would this work for other (non-template) implementation files?

- Yes (36%)
- No (64%)

To prevent confusion, another convention is to use a different extension for the template implementation file.

Examples: .txx, tpp
You still include them at the bottom of the header file.

- This is the convention we will use. See the course FAQ for how to enable highlighting of these files in VS.

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.

Classic 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;
};
```

Classic Shape Example

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 implmentation **on purpose**. 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* (there are other kinds we are ignoring for now). 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:
  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) as

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

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

```
  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

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

You are right! The difference is one between runtime and compile time, or *dynamic* versus *static* polymorphism.

Inheritance versus Composition versus Templates

So, which do you use when?

- Use composition for “is implemented in terms of” or “has a”
- Use inheritance for “is a”
- Use templates for “works with”

In-class Exercise

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 `abstract_bag.hpp` and modified `bag_simple.hpp` files via Canvas at the Assignment "Exercise for Meeting 4".

Next Actions and Reminders

- Read CH pp. 48-66 on Recursion
- Warmup due by noon on Fri 9/8
- Program 0 is due 9/8 by 11:59pm