CS225
Data Structures
Notes Packet #6
Templates

Jason Zych

Chapter 6

Templates

6.1 The basic idea behind templates

Two functions or classes may have the exact same body of code except for a variable type. For example, two sets of linked list code could be defined except one has `int` data in the nodes and one has `char` data in the nodes. Having to write a separate copy of the code for each type you want to use that code with is, among other things, a serious waste of your coding time. It would be nicer if we could use a generic type – i.e. treat the type itself as a variable – and then fill in a more specific type for that generic type variable later on. Templates allow you to do this, and that idea is what we discuss below.

6.2 An example: the swap function

Consider the following code:

```
// integer swap
void swap(int& x, int& y)
{
   int temp;
   temp=x;
   x=y;
   y=temp;
}
```

If in some other function, you had two integer variables, then you could pass those integer variables into the function above to swap their values.

```
int a = 5;
int b = 3;
swap(a, b);  // now b==5 and a==3
```

Now, if you want to have a function that switches values of type `char`, you need to write another `swap` function, overloaded to accept char parameters instead of `int` parameters. But almost all the rest of the code will be the same. Likewise, if you were to write a function to swap two `String` variables, nearly all the code would be the same as it is in the above version. The only thing that would really be different would be the type of the parameters and the type of `temp`. So, let's make those types generic rather than specific.

```
// generic swap
template <class Etype>
void swap(Etype& x, Etype& y)
{
   Etype temp;
   temp=x;
   x=y;
   y=temp;
}
```

Note the use of the `template <class Etype>` line. This line merely says that the following declaration or definition is a template, and that the template variable (the generic type being used) is called "`Etype`". You can use any other variable name you want in place of "`Etype`"; other common template variables are `T` and `GenObj`.

The above would be all you need for the definition of the function. The declaration of the function would be merely the top two lines with a semicolon at the end as with other function declarations. Or in other words, as follows:

```
// declaration for swap
template <class Etype>
void swap(Etype& x, Etype& y);
```

Then, with `swap` properly coded, you can call it many different times in the same code segment. As you can see below, you need to match the pattern of the parameter list for the function, and that parameter list includes both parameters being of the same type, even though that type is indeed generic.

```
      double x, y;
      String s1, s2;
      Coord point1, point2;
      // You'd perform some initializations too,
      // but we won't explicitly show those here.

      swap(x, y);    // x and y are both double
      swap(point1, point2);   // point1 and point2 are Coord objects
      swap(x, s2);    // x is double, s2 is a String  ***ILLEGAL!!
      swap(s1, s2);   // *both* String objects -- this *is* legal
```

To define a class template, we follow a similar procedure. Imagine we have a `Label` class which provides functions to access an integer label.

```
class Label {
   .
   .
   int hiddenLabel;

};
```

If all the functions we want to add will be trying to assign and read the label and things like that – and will not be doing operations that depend on it being an *integer* label (for example, assuming we are not adding labels, which conceptually makes no sense given our idea of what a label is), then we could generalize this type, using the following syntax:

```
template <class T>
class Label {
   .
   .
   T hiddenLabel;

};
```

Once you reach the ending semicolon of the class definition, from that point on there is no class `Label`. The class name, i.e. the type, is *always* `Label<>`, with something filled in those angle brackets. So, when when you declare a class object, you fill in an actual type in those angle brackets:

```
   Label<int>   iLabel;
   Label<char>  bigSticker;
   Label<Coord>*  coordinateLabel = new Label<Coord>();
```

The first line creates a `Label` object with `int` substituted for `T` everywhere `T` occurs in the `Label` class. The second line does the same except `char` is used instead of `int`. And so on.

Since your type is `Label<>` and not `Label`, when you write the class member function definitions outside the class, you need to fill in something else in the angle brackets. Since you don't know the specific type when you first write the code, you must use a generic type, which means putting a template declaration line above the function header. That is, instead of

```
       Label::function1()  { code stuff; }
```

you use the syntax

```
       template <class T>
       Label<T>::function1()  { code stuff; }
```

The only time you can use `Label` instead of `Label<>` is inside the class declaraton itself. If a function had the type `Label` as a parameter type or return type, then using just `Label` instead of `Label<>` in the declaration for that function is fine, since that function declaration would be inside the class declaration. However, once you hit the closing semicolon of the class declaration, then from that point on, you always must follow your use of the type `Label` with the angle brackets filled in with a type. That includes the definitions of any functions that have `Label` as a parameter type or return type. Within the class declaration, the type is `Label`. Outside the class declaration, the type is `Label<some generic or real type>` and that type should be used correctly in all cases.

For example:

```
template<class T>
class Label
{
public:
   // Label is return type and parameter type here
   const Label& operator=(const Label& origVal);
        .
        .
        .
   T hiddenLabel;

};

template <class T>
const Label<T>& Label<T>::operator=(const Label<T>& origVal)
{
   // define function in here
}
```

However, the constructor function itself is not affected by this – because that is a *function* name and not a *type* name. That is, use this:

```
   template <class T>
   Label<T>::Label()
   {
       // constructor code here
   }
```

and NOT this:

```
   template <class T>
   Label<T>::Label<T>()   // WRONG!!!!!!!!!!!!
   {
       // constructor code here
   }
```