6 Software Development with Templates, Iterators, and the STL

Abstract

This chapter aims to provide a better approach to writing code that's meant to be reused. We use templates to write template functions and template classes that are extreme utile. We'll provide iterators for container classes, allowing standard access to the items in a container. By following a standard approach we will write classes that are easier to use and maintain by others, while also taking advantage of the Standard Template Library.

Template Functions

Rather than writing several functions that take different inputs we can simply write one function to handle the different use cases. We can do this by writing one function and a typedef statement like so:

```
typedef ______ item;
item maximal (item a, item b) {
    if (a > b)
        return a;
    else
        return b;
}
```

From here we allow the programmer to fill in any type for the typedef that has a $>$ operator defined and a copy constructor (see pg. 96). However, the typedef approach limits us to one instance of the maximal function. What if we need two?

A more flexible solution is called a **template function**, which is like a function but with one caveat: the definition of a template function depends on an underlying data type. That data type can be defined as *Item* but it's not pinned down to just this specific type. When a template's used, the compiler examines the types of the arguments and automatically determines the datatype of *Item*.

Syntax for a Template Function

```
//Using a typedef statement
typedef int item;
item maximal(item a, item b) {
    if ( a > b)
        return a;
    else
        return b;
```

```
}

//Defining a template function
template <class Item>
Item maximal(Item a, Item b) {
    if ( a > b)
        return a;
    else
        return b;
}
```

The second definition, which uses the template definition, is known as a **tmeplate prefix**. The bracketed code defining *Item* as a class is called the **template parementer**.

Common programming practice is to capitalize the name of template parameters, to differentiate them from specific data types (Int versus int, Item versus item, etc).

An alternative to *class* is *typename*, but *typename* is not supported by older compilers.

Using a Template Function

A program can use a template function with any *Item* type that has the necessary features. In the scope of our previous example, the *Item* definition can be any of the basic data types, or any class that implements a >.

NOTICE: Unification errors regurgitate by the compiler are often the result of a failed template instantiation. Make sure that the template paremeter appears in hte paremeter list of the function.

See pg. 295 for code example.

The ¡algorithm¿ facility containts a swap, max, and min function.

Parameter Matching for Template Functions

The next template function function searches an array for hte biggest item and returns the index.

```
template <class Item>
size_t index_of_maximal(const Item data[], size_t n);
//Precondition: data is an array with at least n items,
//and n > 0.
//Postcondition: The return value is the index of a
//maximal item from data[0]...data[n-1].
```

However this function does not work as intended. The compiler can only do so much guess work on the information it's given. We must be as explicit as possible to ensure the compiler translates our functions properly. The arguments of a template function must therefore be the same.

A Template Function to Find the Biggest Item in an Array

To handle the inconsistency in the previous example we must change the specification slightly.

```
template <class Item, class SizeType>
size_t index_of_maximal(const Item data[], SizeType n);
```

Template Classes

In a similar way that template functions depend on an underlying data type, **template classes** also depends on an underlying data type.

Syntax for a Template Class

The syntax changes slightly between template function and class definitions. We must first change the templace class definition as seen below.

```
//Using a typedef statement
class bag {
    public:
        typedef int value_type;
}

//Using a template class
tmeplate <class Item>
class bag {
    public:
        typedef Item value_type;
}
```

The template expression at the beginning of the second example is a **template prefix**. It notifies the compiler that the following definition will use an unspecified data type. Our next step is to implement functions for the template class, which we can see in the typedef statement of the second example. However, outside the class definition there are a few things we need to do to notify the compiler of the proper data type to use.

- The template prefix is used directly before each function prototype and definition.
- Each use of the class name outside its definition is changed to the template class name (such as bag<Item>). The constructor's name simply remains bag.
- Within a class definition we still use the bag's type names, such as size_type or value_type. (It may be better practice to use Item instead of value_type, for semantic reasons).

– Outside member functions, it is necessary to preface usage with the *typename* keyword: typename bag<Item>::size_type.

```
//Overloading the addition operator for the bag class
template <class Item>
bag<Item> operator+(const bag<Item>& b1, const bag& b2)...

//Another example of template class usage
bag::size_type bag::count(const value_type& target) const...
```

The functions return type is bag::size_type, but it's returned before the compiler defines this as a member function. In order for the above statement to be correct we must add the *typename* keyword.

```
tmeplate <class Item>
typename bag<Item>::size_typ bag<Item>::count
                            (const Item& target) const...
```