Introduction to Templates

Some of the most reusable pieces of code are general container classes, such as arrays, lists, stacks, and binary trees. The major impediment to reusing container classes is having problems dealing with different types. To a compiler, a stack of integers is different from a stack of strings or a stack of airports. What is needed for container classes is **polymorphism**— which is a blindness to the exact type you're dealing with. In theory, a stack behaves the same regardless of what type it holds— it supports `push` and `pop` operations regardless of what type it is holding. There are two main hitches to providing a "stack of any type" in C or C++ (up until today, that is). First, an integer is not necessarily the same size as an airport, so how big do you make each element of a stack if the same stack can hold an airport or an integer? Second, the compiler wants to know what type you're passing to and returning from the stack functions.

On your first assignment, you made use of `SymbolTables` and a `FlightLists` to store airports and flights. These two classes demonstrate two different alternatives for providing container classes — one was polymorphic, one was not.

Readings from Eckel:
Chapter 16: Introducing Templates

Readings from Deitel[2]:
Chapter 12: Templates

Note: Templates have been part of the ANSI standard for four years, but for whatever reason it seems that templates are a hard thing for compilers to do right. Each of the compilers has its own quirky little way of dealing with and expanding templates, and while the template code is likely to be portable from one machine to another, its behavior is not.

Revisiting the `SymbolTable`

As you may recall, the symbol table class we provided was accessed mainly by using the `lookup` and `enter` methods. The methods were defined like this:

```
void enter(const char *key, void *value);
void *lookup(const char *key);
```

This is using the classic C-style polymorphism — also known as the `void *` hack. Basically, C and C++ allow any type of pointer to absorbed into a `void *`, void * is a generic, all-accepting pointer type, so C and C++ programmer can get away with it. The nice thing about doing this is that all pointers are the same size, so we know how big to

make each element in the symbol table. However, the bad thing is that you surrender all type checking ability when doing this. For example, if you were storing airports in the symbol table, you could do this:

```
Airport* airport = new Airport("SFO", "San Francisco, CA");
double* doublePtr = new double(3.14159);
airports.enter("SFO", airport);
airports.enter("JFK", doublePtr);
```

Later, when you go to `lookup` "JFK" thinking that it's an airport, you've got some surprises coming! You didn't really want a single symbol table object to hold different types. What you really wanted was one symbol table object to hold **just** integers, and another symbol table object to hold **just** airports, etc.

Revisiting the `FlightList`

The `FlightList` had plenty of type checking, but it's big problem was that it only held flights. You accessed the `FlightList` with such methods as:

```
Flight *firstFlight();
void addFlight(Flight *newElem);
```

You'd never make the mistake of putting a pointer to a `double` in the `FlightList` . The problem is if you wanted an `IntegerList`, or an `AirportList`, you'd have to copy all of the code for the class, and change every reference to `Flight` to whatever type you were trying to hold in the list. This quickly becomes very tedious and error-prone.

Templates — The Best of Both Worlds

What we really want is to have the best of both worlds for our container classes. We want good type checking, and we don't want to have to write a version of the container class for each type we want to store. C++'s answer to this problem is the template—templates provide a way to parameterize one or more types in a function or a class. The compiler will make sure that your types match, and you only have to write the template once.

A Simple `StringQueue`

Let's revisit what I assume is a familiar abstraction from CS106B or CS106X: the Queue. Rather than make the abstraction fully polymorphic in the traditional C manner, lets work with a specialized version of a queue that stores only `string`s. While examining the code, understand that the queue abstraction, like many other abstractions, is that of a container class—it in no way depends on the functionality of the `string` class. In fact, the only detail the `Queue` class needs at compile-time is exactly how big a `string` is; that's it!

```
class Queue
{
   public:
      Queue();
      ~Queue();

      bool empty() const;
      void enqueue(const string& str);
      string dequeue();
      const string & peek() const;
   private:
      struct Node {
         string str;
         struct Node *next;
         Node(const string& str, struct Node *next) : str(str), next(next);
      } *head, *tail;
};
```

The constructors and destructors would look like this:

```
Queue::Queue()
{
   head = tail = NULL;
}

Queue::~Queue()
{
   Node *next;
   while (head != NULL) {
      next = head->next;
      delete head; // doesn't delete the next field recursively
      head = next;
   }
}
```

The or interesting code, however, presents itself within the `enqueue`, `dequeue`, and `peek` methods.

```
void Queue::enqueue(const string& str)
{
   Node *newTail = new Node(str, NULL);
   if (head == NULL)
      head = newTail;
   else
      tail->next = newTail;

   tail = newTail;
}

string Queue::dequeue()
{
   assert (head != NULL);

   string str = head->str;
   Node *old = head;
   head = head->next;
   if (head == NULL) tail = NULL;  // make sure tail knows list is empty
```

```
   delete old;
   return str;
}

const string& Queue::peek() const;
{
   assert (head != NULL);
   return head->str;
}
```

A Simple Templatized Version of our `Queue`

To rewrite the above code for `Queue`s containing different types would require that we either cut and paste code to another file, or to generalize the above abstraction to store `void` pointers.

```
template <class Element>
class Queue {
   public:
      Queue();
      ~Queue();

      bool empty() const;
      void enqueue(const Element& elem);
      Element dequeue();
      const Element& peek() const;

   private:
      struct Node {
         Element elem;
         struct Node *next;
         Node(const Element& elem, struct Node *next) :
                                          elem(elem), next(next) {}
      } *head, *tail;
};
```

The definition of the class uses a template parameter, which in this case is the element type. A template parameter is often, but doesn't have to be, the element type of a container. In this example, `class` is the parameter type, and `Element` is the parameter name. Template parameters can be other types, such as integers. You can also have more than one template parameter. Within the class definition, we can use `Element` to represent our "generic" contained type. Really, this is a placeholder which gets filled in (at compile-time) when we instantiate a `Queue` object. We have to use the template specifier when we implement the `Queue` member functions. The constructor would look like this:

```
template <class Element>
Queue<Element>::Queue()
{
   head = tail = NULL;
}
```

The more involved enqueue and dequeue operation would look like this:

```
template <class Element>
void Queue<Element>::enqueue(const string& elem)
{
   Node *newTail = new Node(elem, NULL);
   if (head == NULL)
      head = newTail;
   else
      tail->next = newTail;

   tail = newTail;
}

template <class Element>
Element Queue<Element>::dequeue() throw (EmptyQueueException)
{
   assert(head != NULL);

   Element elem = head->elem;
   Node *old = head;
   head = head->next;
   if (head == NULL) tail = NULL;  // make sure tail knows list is empty

   delete old;
   return elem;
}
```

Instantiating Templates

To instantiate a template, you simply declare an Queue like you normally would, save the fact that you must specify exactly what data type should be stored in the Queue. Once the compiler sees this data type, it generates code on your behalf and compiles it as if you hand hand-typed the specialized Queue type in yourself.

```
int main()
{
   Queue<int> intQueue;
   Queue<char> charQueue;

   for (int i = 0; i < 26; i++) {
      intQueue.enqueue(i);
      charQueue.enqueue('a' + i);
   }

   while (!intQueue.empty()) {
      cout << setw(2) << intQueue.dequeue() << " " << charQueue.dequeue() << endl;
   }
}
```

Template Functions

Global functions can be used with templates as well. This is useful for utility functions that can apply to any type, as well as for implementing user-defined operators for classes. Suppose we wanted to write a `max` function which would `return` the maximum of its two arguments. We could write this like this:

```
template <class Type>
inline Type max(Type value1, Type value2)
{
   if (value1 > value2)
      return value1;
   else
      return value2;
}
```

Note that every type we use with this function must provide an `operator>` for this function to work. If there is no `operator>` for the type provided, there will be a compile-time error. We can simply call the `max` function like this:

```
int intMax = max(0, 1);
double doubleMax = max<double>(0.1, 1.0);
string str1("Hi"), str2("There");
string strMax = max(str1, str2);
```

We don't have to specify the template argument in these cases because the compiler figures it out for us. However, we could provide the template argument it if we wanted to, and we have to provide it if the compiler can't figure it out otherwise.

Specialization

The `max` code brings up an interesting point. Consider comparing two C-strings with the `max` function. If we wrote:

```
char *strMax = max("Hi", "There");
```

We would get an undefined result. This is because `operator>(const char *, const char *)` is performing a pointer-versus-pointer comparison on the two addresses we pass it. We don't really know or care which string is at in memory. We are more concerned about the lexicographic ordering of the two strings. The templatized function does one thing (which is usually right), but in this case we want a different flavor of `max(const char *, const char *)`. Enter, stage left: template specialization!

When the generic source code of a template isn't exactly what we want, we can override the template behavior. In doing so, we need to specify exactly what type the specialization is designed for (using the same template syntax while specifying an actual type instead of a placeholder) and then provide the specialized implementation. Effectively, we are providing the routine as if the compiler had generated it from the

template. When the compiler goes to generate the code for a function, it uses the code we provided instead.

In order to provide a specialized version of the `max` function for strings, we need to provide the routine like this:

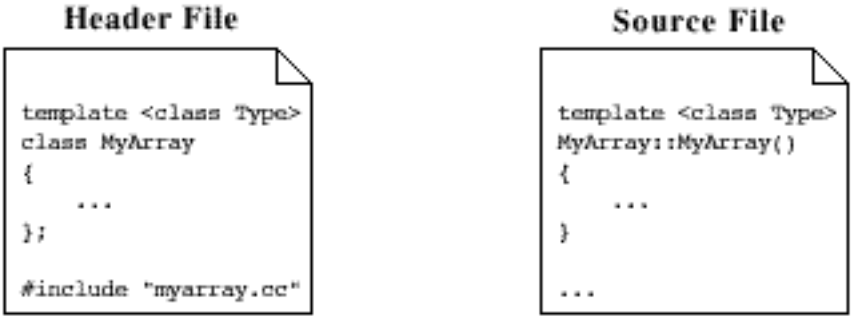
```
const char *max(const char *str1, const char *str2)
{
   if (strcmp(str1, str2) > 0)
      return str1;
   else
      return str2;
}
```

Now if we ever call `max` with strings, the specialized version will be called.

Where to Put Code

Templates work a bit different from other forms of C++ code. Instead of compiling the template code into a single object file, and then re-using that code, generally what happens is that a new copy of the code is generated for each different template argument used. This means that if you have a `Queue` of integers and a `Queue` of `strings`, you have two copies of the `Queue` code. Or, if you called `max` on two integers, and `max` on two `strings`, you would get two copies of the `max` code.

Because the compiler needs to generate separate copies of the template function or template class code, you don't simply add the `.cc` file to your project or makefile. Instead, you `#include` the proper header file which should contain both the class or function definition and the implementation. I generally structure my templates like this



to make them a little bit cleaner:

I do this because I don't like to put the implementation of any classes or functions in a header file. The template still represents an abstraction, so in the spirit of abstraction and information hiding, I place all the implementation code in a `.cc` file as if it were a normal `class`.