TEMPLATES

1 Templates

We have seen that functions can take arguments of specific types and have a specific return type. We now consider *templates,* which allow us to work with *generic types.* Through templates, rather than repeating function code for each new type we wish to accommodate, we can create functions that are capable of using the same code for different types. For example:

```
int sum(const int x, const int y) {
    return x + y;
}
```

For this function to work with doubles, it must be modified to the following:

```
double sum (const double x, const double y) {
    return x + y;
}
```

For a simple function such as this, it may be a small matter to just make the change as shown, but if the code were much more complicated, copying the entire function for each new type can quickly become problematic. To overcome this we rewrite `sum` as a function template.

The format for declaring a function template is:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

Both forms are equivalent to one another, regardless of what type *identifier* ends up being. We can then use *identifier* to replace all occurrences of the type we wish to generalize.

So, we rewrite our `sum` function:

```
template <typename T>
T sum(const T a, const T b) {
    return a + b;
}
```

Now, when `sum` is called, it is called with a particular type, which will replace all `T`s in the code. To invoke a function template, we use:

```
function_name <type> (parameters);
```

Here is an example `main` function using the above `sum` function template:

```
int main() {
    cout << sum<int>(1, 2) << endl;
    cout << sum<float>(1.21, 2.43) << endl;
    return 0;
}
```

This program prints out `3` and `3.64` on separate lines.

The *identifier* can be used in any way inside the function template, as long as the code makes sense after *identifier* is replaced with some type.

It is also possible to invoke a function template without giving an explicit type, in cases where the generic type *identifier* is used as the type for a parameter for the function. In the above example, the following would also have been valid:

```
int main() {
    cout << sum(1, 2) << endl;
    cout << sum(1.21, 2.43) << endl;
    return 0;
}
```

Templates can also specify more than one type parameter. For example:

```
#include <iostream>
using namespace std;

template <typename T, typename U>
U sum(const T a, const U b) {
    return a + b;
}

int main() {
    cout << sum<int, float>(1, 2.5) << endl;
    return 0;
}
```

This program prints out `3.5`. In this case we can also call sum by writing `sum(1, 2.5)`.

Class templates are also possible, in much the same way we have written function templates:

```
#include <iostream>
using namespace std;

template <typename T>
class Point {
   private:
         T x, y;
   public:
         Point(const T u, const T v) : x(u), y(v) {}
         T getX() { return x; }
         T getY() { return y; }
};

int main() {
    Point<float> fpoint(2.5, 3.5);
    cout << fpoint.getX() << ", " << fpoint.getY() << endl;
    return 0;
}
```

The program prints out `2.5, 3.5`.

To declare member functions externally, we use the following syntax:

```
template <typename T>
T classname<T>::function_name()
```

So, for example, getX could have been declared in the following way:

```
template <typename T>
T Point<T>::getX() { return x; }
```

assuming a prototype of `T getX();` inside the class definition.

We can also define different implementations for a single template by using *template specialization.* Consider the following example:

```
1  #include <iostream>
2  #include <cctype>
3  using namespace std;
4
5  template <typename T>
6  class Container {
7     private:
8           T elt;
9     public:
10          Container(const T arg) : elt(arg) {}
11          T inc() { return elt+1; }
12 };
13
14 template <>
15 class Container <char> {
16    private:
17          char elt;
18    public:
19          Container(const char arg) : elt(arg) {}
20          char uppercase() { return toupper(elt); }
21 };
22
23 int main() {
24    Container<int> icont(5);
25    Container<char> ccont('r');
26    cout << icont.inc() << endl;
27    cout << ccont.uppercase() << endl;
28    return 0;
29 }
```

This program prints out `6` and `R` on separate lines. Here, the class `Container` is given two implementations: a generic one and one specifically tailored to the `char` type. Notice the syntax at lines 14 and 15 when declaring a specialization.

Finally, it is possible to parametrize templates on regular types:

```
#include <iostream>
using namespace std;

template <typename T, int N>
class ArrayContainer {
   private:
         T elts[N];
   public:
         T set(const int i, const T val) { elts[i] = val; }
         T get(const int i) { return elts[i]; }
};

int main() {
   ArrayContainer <int, 5> intac;
   ArrayContainer <float, 10> floatac;
   intac.set(2, 3);
   floatac.set(3, 3.5);
   cout << intac.get(2) << endl;
   cout << floatac.get(3) << endl;
```

```
  return 0;
}
```

This program prints out `3` and `3.5` on separate lines. Here, one instance of the ArrayContainer class works on a 5-element array of `int`s whereas the other instance works on a 10-element array of `float`s.

Default values can be set for template parameters. For example, the previous template definition could have been:

```
template <typename T=int, int N=5> class ArrayContainer { ... }
```

and we could have created an `ArrayContainer` using the default parameters by writing:

```
ArrayContainer<> identifier;
```

2 Standard Template Library

Part of the C++ Standard Library, the *Standard Template Library* (STL) contains many useful container classes and algorithms. As you might imagine, these various parts of the library are written using templates and so are generic in type. The containers found in the STL are lists, maps, queues, sets, stacks, and vectors. The algorithms include sequence operations, sorts, searches, merges, heap operations, and min/max operations. We will explore how to use some of these through example here:

```
1   #include <iostream>
2   #include <set>
3   #include <algorithm>
4   using namespace std;
5
6   int main() {
7     set<int> iset;
8     iset.insert(5);
9     iset.insert(9);
10    iset.insert(1);
11    iset.insert(8);
12    iset.insert(3);
13
14    cout << "iset contains:";
15    set<int>::iterator it;
16    for(it=iset.begin(); it != iset.end(); it++)
17          cout << " " << *it;
18    cout << endl;
19
20    int searchFor;
21    cin >> searchFor;
22    if(binary_search(iset.begin(), iset.end(), searchFor))
23          cout << "Found " << searchFor << endl;
24    else
25          cout << "Did not find " << searchFor << endl;
26
27    return 0;
28  }
```

In this example, we create an integer set and insert several integers into it. We then create an iterator corresponding to the set at lines 14 and 15. An iterator is basically a pointer that provides a view of the set. (Most of the other containers also provide iterators.) By using this iterator, we display all the elements in the set and print out `iset contains: 1 3 5 8 9`. Note that the set automatically sorts its own items. Finally, we ask the user for an integer, search for that integer in the set, and display the result.

Here is another example:

```
#include <iostream>
#include <algorithm>
using namespace std;

void printArray(const int arr[], const int len) {
   for(int i=0; i < len; i++)
         cout << " " << arr[i];
   cout << endl;
}

int main() {
   int a[] = {5, 7, 2, 1, 4, 3, 6};

   sort(a, a+7);
   printArray(a, 7);
   rotate(a,a+3,a+7);
   printArray(a, 7);
   reverse(a, a+7);
   printArray(a, 7);

   return 0;
}
```

This program prints out:

```
 1 2 3 4 5 6 7
 4 5 6 7 1 2 3
 3 2 1 7 6 5 4
```

The STL has many, many more containers and algorithms that you can use. Read more at http://www.cplusplus.com/reference/stl and http://www.cplusplus.com/reference/algorithm/.

3 Operator Overloading

We have been using operators on primitives, but sometimes it makes sense to use them on user-defined datatypes. For instance, consider the following struct:

```
struct USCurrency {
      int dollars;
      int cents;
};
```

Perhaps we would like to add two `USCurrency` objects together and get a new one as a result, just like in normal addition:

```
USCurrency a = {2, 50};
USCurrency b = {1, 75};
USCurrency c = a + b;
```

This of course gives a compiler error, but we can define behavior that our datatype should have when used with the addition operator by overloading the addition operator. This can be done either inside the class as part of its definition (the addition from the point of view of the object on the left side of the `+`):

```
USCurrency operator+(const USCurrency o) {
   USCurrency tmp = {0, 0};
   tmp.cents = cents + o.cents;
   tmp.dollars = dollars + o.dollars;
```

```
    if(tmp.cents >= 100) {
          tmp.dollars += 1;
          tmp.cents -= 100;
    }

    return tmp;
}
```

or outside the class as a function independent of the class (the addition from the point of view of the `+`):

```
USCurrency operator+(const USCurrency m, const USCurrency o) {
    USCurrency tmp = {0, 0};
    tmp.cents = m.cents + o.cents;
    tmp.dollars = m.dollars + o.dollars;

    if(tmp.cents >= 100) {
          tmp.dollars += 1;
          tmp.cents -= 100;
    }

    return tmp;
}
```

Similarly, we can overload the `<<` operator to display the result:

```
ostream& operator<<(ostream &output, const USCurrency &o)
{
    output << "$" << o.dollars << "." << o.cents;
    return output;
}
```

Assuming the above definitions, we can run the following program:

```
int main() {
   USCurrency a = {2, 50};
   USCurrency b = {1, 75};
   USCurrency c = a + b;
   cout << c << endl;
   return 0;
}
```

and get the printout `$4.25`.

The list of overloadable operators:

+	-	*	/	+=	-=	*=	/=	%	%=	++	--
=	==	<	>	<=	>=	!	!=	&&	||		
<<	>>	<<=	>>=	&	^	|	&=	^=	|=	~	
[]	()	,	->*	->	new	new[]	delete		delete[]		

Source:http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/lecture-notes/MIT6_096IAP11_lec09.pdf