C++ Templates

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/templates/

Templates

Templates are a way to parameterize your variable types.

What does that mean?

Templates

When we define functions, we're specifying some kind of common set of operations we want to do for any input of a given type, eg:

```
int get_max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Templates

Unfortunately, this function only works for inputs of type int (or of types that can automatically become int). This limits its usefulness.

What if we wanted to write a get_max function which could work for any type? This would prevent us from having to cut and paste the get_max function for every type we wanted to use it for.

```
int get_max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Function Templates

We can use templates to specify (at compile time) what types are used in a function. The compiler will generate a copy of the function for each type the function is called with for us.

```
template <class T>
T get_max (T a, T b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Note this is unlike Java's generics (even though they look similar and are used for similar purposes). Java's generics do not actually create copies of the functions and are only used by the compiler for type checking.

Function Templates

Note that the compiler only generates copies of the template function when those types are specified. (This can be a headache in multiple file projects, but more on that later).

```
template <class T>
T get_max (T a, T b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

int main() {
   int a = 5, b = 10;
   cout << “the max of a and b is: “ << get_max<int>(a, b) << endl;

   double c = 3.5, d = 35.1;
   cout << “the max of c and d is: “ << get_max<double>(c, d) << endl;
}
```

Function Templates

In this case, the compiler would have generated this code:

```
int get_max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

```
double get_max (double a, double b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

int main() {
   int a = 5, b = 10;
   cout << “the max of a and b is: “ << get_max<int>(a, b) << endl;

   double c = 3.5, d = 35.1;
   cout << “the max of c and d is: “ << get_max<double>(c, d) << endl;
}
```

Function Templates

So taking the following template:

```
template <class T>
T get_max (T a, T b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

And invoking it with the template argument int.

```
int a = 5, b = 10;
cout << “the max of a and b is: “ << get_max<int>(a, b) << endl;
```

Creates the function with all T replaced with int. In this way, types can be parameters to functions as well. That’s what I meant by parameterize types.

```
int get_max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Function Templates

As all the compiler does is replace the templates name with the passed types, you can do anything that would be allowable normally by those types, eg:

```
template <class T>
T get_max (T a, T b) {
  T result;
  if (a > b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}
```

So you can create new instances of a template variable (unlike in Java).

Function Templates

You can also use multiple classes in your templates:

```
template <class T1, class T2>
void get_max2(T1 a, T1 b, T1 &max1, T2 c, T2 d, T2 &max2) {
  max1 = (a > b)? a : b;
  max2 = (c > d)? c : d;
}
```

Inferring Types

In many cases, the compiler can infer the types from your template so you don't need to specify them:

```
template <class T>
T get_max (T a, T b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

int main() {
  int a = 5;
  int b = 7;
  //The compiler knows it can use the type int for the template.
  cout << "the max of a and b is: " << get_max(a,b) << endl;

  int c = 10;
  long d = 15;
  //The compiler can't know if you want to use int or long for
  //the template so it will spit out an error:
  cout << "the max of c and d is : " << get_max(c,d) << endl;
}
```

Non-Type Templates

You can also template values instead of types. This can lead to performance improvements in some cases, because the value can be filled in at compile time (instead of being passed as a parameter).

```
template <int N, class T>
T sum_first(T *a) {
  T sum;
  for (int i = 0; i < N; i++) {
    sum += a[i];
  }
 return sum;
}

int main() {
  int array = {3, 2, 10, 22, -5, 32 };

  //The compiler can fill in 3 for the for loop, and with
  //compiler optimizations it can unroll that loop:
  cout << “the sum of the first 3 values is: “
       << sum_first<3,int>(array) << endl;
}
```

Non-Type Templates

This would create a function that looked like:

```
int sum_first(int *a) {
  int sum;
  for (int i = 0; i < 3; i++) {
    sum += a[i];
  }
 return sum;
}

int main() {
  int array = {3, 2, 10, 22, -5, 32 };

  //The compiler can fill in 3 for the for loop, and with
  //compiler optimizations it can unroll that loop:
  cout << “the sum of the first 3 values is: “
       << sum_first<3,int>(array) << endl;
}
```

Non-Type Templates

Which most compilers will optimize to:

```
int sum_first(int *a) {
  int sum;
  sum += a[0];
  sum += a[1];
  sum += a[2];
  return sum;
}

int main() {
  int array = {3, 2, 10, 22, -5, 32 };

  //The compiler can fill in 3 for the for loop, and with
  //compiler optimizations it can unroll that loop:
  cout << “the sum of the first 3 values is: “
       << sum_first<3,int>(array) << endl;
}
```

Class Templates

It is also possible to apply templates to classes. For example, all your tree classes would only work for one type. This can be fixed:

```
template<class T>
class TreeNode {
   private:
      TreeNode<T> *left;
      TreeNode<T> *right;
      T value
   public:
      TreeNode(T v) : value(v) {}
};

template<class T>
class Tree {
   private:
      TreeNode<T> *root;
   public:
      …
}

int main() {
   Tree<string> my_tree = new Tree<string>();
}
```

Template Class Methods

To define member functions of template classes, you need to continue to specify the template for each definition:

```
template<class T>
class TreeNode {
   private:
      TreeNode<T> *left;
      TreeNode<T> *right;
      T value
   public:
      TreeNode(T v) : value(v) {}
      void set_right(TreeNode<T>*);
      void set_left(TreeNode<T>*);
};

template<class T>
void TreeNode<T>::set_right(TreeNode<T> *r) {
  right = r;
}

template<class T>
void TreeNode<T>::set_left(TreeNode<T> *l) {
  left = l;
}
```

Template Specialization

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

In some cases, you might want a different definition of a class or function for a particular template type:

Template Specialization

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

Note that this is defined differently, given the templated class:

```
template <class T> class mycontainer { ... };
```

The specialization is:

```
template <> class mycontainer <char> { ... };
```

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

Notes on Multiple File Projects

If you have a header file (.hxx) and a source file (.cxx) which has a template and it's definitions, and want to use that template in another .cxx file by including the header — the templates must have been created in the source .cxx file.

This can be a big pain, but you can somewhat fix it by making definitions of the template in the source file that don't do anything. (This is a longstanding problem with C++).