Templates

Reid Watson
(rawatson@stanford.edu)

Administrivia

- Assignment one is still out!
- LaIR hours 8-12 tonight!

Why Templates?

I want to write a function to find the minimum of two `int`s.

Why Templates?

Solution 1: The obvious way

Here's a nice simple solution to this problem.

```
int min(int a, int b) {
   return (a < b) ? a : b;
}
```

Why Templates?

What will happen if I try and use this function with doubles?

```
double x = 1.0;
double y = 2.5

double smaller = min(x,y);
```

Why Templates?

Solution 2: The C version

In C, this problem was traditionally solved by writing multiple versions of the function and giving them different names:

```
int minint(int a, int b) {
   return (a < b) ? a : b;
}
double mindouble(double a, double b) {
   return (a < b) ? a : b;
}
```

Why Templates?

Problems: This is terrible!

- We now have to write the type of our variables when we want when we use min
- We have to type out multiple copies of the same function
- Creating a new type means writing a new min function for it.

Why Templates?

Solution 3: Function Overloading

Function overloading allows a C++ programmer to define multiple functions with the same name but different parameter types.

```
int min(int a, int b) {
   return (a < b) ? a : b;
}
double min(double a, double b) {
   return (a < b) ? a : b;
}
```

Why Templates?

Problems: This is (still) terrible!

- ~~We now have to write the type of our variables when we want when we use min~~
- We have to type out multiple copies of the same function
- Creating a new type means writing a new min function for it.

Why Templates?

Solution 4: Templates!

Templates allow us to use the same function on variables of any type.

```
template<typename T>
T min(T a, T b) {
   return (a < b) ? a : b;
}
```

What is a Template?

- A **template function** defines a blueprint for generating functions.
- It's equivalent to having the compiler automatically write out every instance of min you need for every type you need.
- **Template instantiation** occurs when a template function is used for a specific type of variable

What is a Template?

To declare a template function, just add the following line before the function definition:

```
template <typename T>
```

'T' is the **template parameter**, which will be replaced with a specific type when you use the template function

What is a Template?

Using a template function is simple. The expression `min<double>` indicate that we want to instantiate and use the min function template for doubles.

```
double x = 1.0;
double y = 2.5

double smaller = min<double>(x,y);
```

What is a Template?

We can actually omit the `<double>` bit here, as the template type can be **inferred**. The compiler notices that both of the arguments are doubles, so it knows that we must have the template parameter T equal to double.

```
double x = 1.0;
double y = 2.5

double smaller = min(x,y);
```

What is a Template?

The compiler will "verify" the instantiation of a template function.

Template functions will only work if every operation used on a variable of template type is supported by the type being instantiated.

What is a Template?

```
#include <iostream>
#include <vector>
using namespace std;
template<typename T>
void print(T& x) {
  cout << x << endl;
}

int main() {
  vector<int> v;
  print(v);
}
```

- This will fail to compile.
- You cannot instantiate the print function template with template parameter T = vector<string>
- This happens because you can't print a vector to cout using '<<', so you can't create a version of print for vectors.

What is a Template?

- Unfortunately, the error message for our last example can be a bit incomprehensible.
- Fitting it onto one slide would require a font smaller than the minimum sized font on Google docs...
- 114 lines of error messages from one line of code

What is a Template?

In general, the first part of the error is the most important:

```
test.cc: In instantiation of ‘void print(T&) [with T = std::
vector<int>]’:

test.cc:11:10:   required from here

test.cc:6:3: error: no match for ‘operator<<’ in ‘std::cout <<
x’
```

This tells us that on line 11, template instantiation of print failed because "operator<<" (printing to an output stream) wasn't supported on our template type.

Using Templates in Algorithm

That's great, but we do we need templates in the STL?

Using Templates in Algorithms

- Let's talk about the "iterator type problem"
- If I call begin or end on a vector of ints, I get an iterator with type vector<int>::iterator.
- This means that it's an iterator for a vector of ints.
- I don't really care about either of those things when I'm writing an algorithm -- the whole point is that I don't want to have to worry about what's behind my iterator!

Using Templates in Algorithms

- Templates are used to solve this problem
- By writing a function template, we can let the compiler do the hard work of generating a version for every different type of iterator
- Let's take a look at how we can use function templates to solve our "iterator problem"

Using Templates in Algorithms

Let's implement a version of the find function in find.cpp

Iterator Types

Up until now I've been referring to iterators as if all iterators were the same.

I lied!

Iterator Types

All iterators can be created using an existing iterator and can be advanced using ++

```
vector<int> v;
v.push_back(1);
v.push_back(2);
vector<int>::iterator i = v.begin();
i++;
++i;
```

Iterator Types

Input iterators can be dereferenced on the *right* hand side of an expression:

```
vector<int> v;
v.push_back(1);
vector<int>::iterator i = v.begin();
int first_element = *i;
```

Iterator Types

Output iterators can be dereferenced on the *left* hand side of an expression:

```
vector<int> v;
v.push_back(1);
vector<int>::iterator i = v.begin();
*i = 2;
```

Iterator Types

Bidirectional iterators can be decremented using --.

```
vector<int> v;
v.push_back(1);
v.push_back(2);
vector<int>::iterator i = v.end();
i--;
--i;
```

Iterator Types

Random Access Iterators can be incremented or decremented by arbitrary amounts using +,- and friends.

```
vector<int> v;
v.push_back(1);
v.push_back(2);
vector<int>::iterator i = v.end();
i = i + 1;
i -= 1;
```

Algorithms Using Iterator Types

We talked a lot about the copy function last class.

Let's peek at the definition of the copy function first to get an idea of what the syntax looks like.

See code in copy.cpp

Algorithms Using Iterator Types

Now let's take a look at how the compiler verifies this function...

Algorithms Using Iterator Types

Template Types	Parameters

`typename InputIterator`
`typename OutputIterator`
	`InputIterator first`
	`InputIterator last`
	`OutputIterator result`

```
while (first != last) {
   *result = *first;
   ++result;
   ++first;
}
return result;
```

Algorithms Using Iterator Types

Template Types	Parameters

`typename InputIterator`
`typename OutputIterator`
	`InputIterator first`
	`InputIterator last`
	`OutputIterator result`

```
while (first != last) {
   *result = *first;
   ++result;
   ++first;
}
return result;
```

Algorithms Using Iterator Types

Template Types	Parameters

`typename InputIterator`
`typename OutputIterator`
	`InputIterator first`
	`InputIterator last`
	`OutputIterator result`

```
while (first != last) {
    *result = *first;
    ++result;
    ++first;
}
return result;
```

Writing Our Own <algorithm>

Let's try writing a version of copy which will only copy certain elements.

See code in copy_if.cpp