Templates and Iterators

...

Original from : Mike Precup (mprecup@cs.stanford.edu)
ENJMIN Edition 2016

Onward and Upward

We've covered:

- Streams
- Containers
- The Basics of Iterators

Templates?

The Problem

Minimum Function

Here's a quick one-liner for finding the minimum of two ints:

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

Minimum Function

Here's a quick one-liner for finding the minimum of two ints:

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


min(5, 3); //3
```

Minimum Function

Here's a quick one-liner for finding the minimum of two ints:

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


min(5, 3); //3
min(6.7, 9.5); //6?
```

Making the Perfect Copy Pasta

In typical C fashion, we could write two functions with different names:

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

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

Three's a Crowd

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

double min_double(double a, double b) {
    return (a < b) ? a : b;
}

size_t min_sizet(size_t a, size_t b) {
    return (a < b) ? a : b;
}
```

Uh Oh

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

double min_double(double a, double b) {
    return (a < b) ? a : b;
}

size_t min_sizet(size_t a, size_t b) {
    return (a < b) ? a : b;
}

float min_float(float a, float b) {
    return (a < b) ? a : b;
}
```

Has Science Gone Too Far?

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

double min_double(double a, double b) {
    return (a < b) ? a : b;
}

size_t min_sizet(size_t a, size_t b) {
    return (a < b) ? a : b;
}

float min_float(float a, float b) {
    return (a < b) ? a : b;
}

char min_char(char a, char b) {
```

The Problem

Multiple copies of the same function for different types

The Problems

Multiple copies of the same function for different types

You have to write out the type name whenever you use it

The Problems

Multiple copies of the same function for different types

You have to write out the type name whenever you use it

If you add a new type, you need a new function

More Overloaded Than My Calendar

Function Overloading lets us have multiple functions with the same name in C++!

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

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

They need different parameters, though.

And we’re good to go!

Nope, Still Bad

Multiple copies of the same function for different types

~~You have to write out the type name whenever you use it~~

If you add a new type, you need a new function

Go Go Gadget Templates!

Templates let you use the same function for a variety of types

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

What is a Template?

We made our min functions through a set of rules:

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

To turn our int min function into one that worked on doubles, we replaced each instance of int with double:

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

What is a Template?

We can give those rules to the compiler in the form of a **template function** by telling it what needs to get replaced. Just before the function, we specify a **template parameter**.

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

It will replace that parameter for us!

What is a Template?

Whenever the compiler sees you use that function for a new type, it will generate a new function with the correct type. This is called **template instantiation**.

Using a Template

To use a template function, you can just call it like any other function. You can indicate the type through angle brackets after the function name, like so:

```
int a = 4, b = 9;
int c = min<int>(a, b);

double a = 4.5, b = 9.2;
double c = min<double>(a, b);
```

Two Steps Forward, One Step Back

~~Multiple copies of the same function for different types~~

You have to write out the type name whenever you use it

~~If you add a new type, you need a new function~~

Using a Template

However, if the type can be **inferred**, you don't actually need to use the angle brackets. If every argument of type T has the same type, the compiler will figure it out.

```
int a = 4, b = 9;
int c = min(a, b);

double a = 4.5, b = 9.2;
double c = min(a, b);
```

Using a Template

Up until now, I've been using T as the template parameter, but you can use whatever you want!

Generally, a more descriptive name is better, but you'll frequently see T, TT, and Z.

Templates in Action

Let’s use templates to write a basic println function like the one in Java.

Print.cpp

When Things Go Wrong

Any time template instantiation occurs, the compiler will check that all the operations used on the templatized type are supported by that type. Let’s throw a vector in, just to see the havoc.

Print.cpp

Thanks, Bjarne

“C makes it easy to shoot yourself in the foot;

C++ makes it harder, but when you do it blows your whole leg off.”

-Our good friend Bjarne

Errors

The first part of the error is usually the most important. In our vector printing example, these were the first lines:

```
..\Print\main.cpp: In instantiation of 'void println(T) [with T = std::vector<int>]':
..\Print\main.cpp:11:15: required from here
..\Print\main.cpp:6:14: error: no match for 'operator<<' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and  'std::
vector<int>')
        std::cout << t << std::endl;
```

From just the first few lines, we know that template instantiation for println failed on line 11, because ‘operator<<’ wasn’t supported for the type we supplied.

Templates ft. Iterators

There's a different type of iterator for every collection:

```
vector<int> v;
vector<int>::iterator itr = v.begin();


vector<double> v;
vector<double>::iterator itr = v.begin();


deque<int> v;
deque<int>::iterator itr = v.begin();
```

Templates ft. Iterators

The whole point of iterators is that I shouldn’t have to worry about that! Iterators are a standard interface to data, but we still had to specify what data the iterator was pointing to.

Now we don’t! The compiler can generate everything we need from a template function.

Templates ft. Iterators

So what can we do with this new combination? Let’s find out!

Find.cpp

Concepts

```
#include <string>
#include <locale>
using namespace std::literals;

// Declaration of the concept "EqualityComparable", which is satisfied by
// any type T such that for values a and b of type T,
// the expression a==b compiles and its result is convertible to bool
template<typename T>
concept bool EqualityComparable = requires(T a, T b) {
    { a == b } -> bool;
};

void f(EqualityComparable&&); // declaration of a constrained function template
// template<typename T>
// void f(T&&) requires EqualityComparable<T>; // long form of the same

int main() {
  f("abc"s); // OK, std::string is EqualityComparable
  f(std::use_facet<std::ctype<char>>(std::locale{})); // Error: not EqualityComparable
```

Iterator Types

Wait, types? Haven’t we been saying iterators are interchangeable?

Woops.

Sharing is Caring

All iterators share a few common traits:

- They can be created from an existing iterator
- They can be advanced using ++
- They can be compared with == and !=

```
vector<double> v = ...;
vector<double>::iterator itr = v.begin();
for (; itr != v.end(); itr++) {
    //...
}
```

Iterator Types

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

```
vector<double> v = ...;
vector<double>::iterator itr = v.begin();
double val = *itr;
```

Iterator Types

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

```
vector<double> v = ...;
vector<double>::iterator itr = v.begin();
*itr = 2.5;
```

Iterator Types

Random access iterators can be incremented or decremented by arbitrary amounts using +, -, and related operations:

```
vector<int> v = ...;
vector<int>::iterator itr = v.begin();
for (; itr != v.end(); itr += 3) {
    //...
}
```

Using Templates and Iterators

Let’s write some code to copy the elements of one collection to another

Copy.cpp

Using Templates and Iterators

Let’s write some code to copy the elements of one collection to another conditionally

CopyIf.cpp

Implicit Interface

What types are valid for a template? Any that satisfy its **implicit interface**.

Implicit Interface

```
template <typename T>
int foo(T input) {
   int i;
   if (input >> i && input.size() > 0) {
      input.push_back(i);
      return i;
   } else {
      return 5;
   }
}
```

Implicit Interface

```
template <typename T>
int foo(T input) {
    int i;
    if (input >> i && input.size() > 0) {
        input.push_back(i);
        return i;
    } else {
        return 5;
    }
}
```

```
input >> int
input.size()
input.push_back(int)
```

Implicit Interface

Basically: if you replaced all the `Ts` with that typename, would it compile?