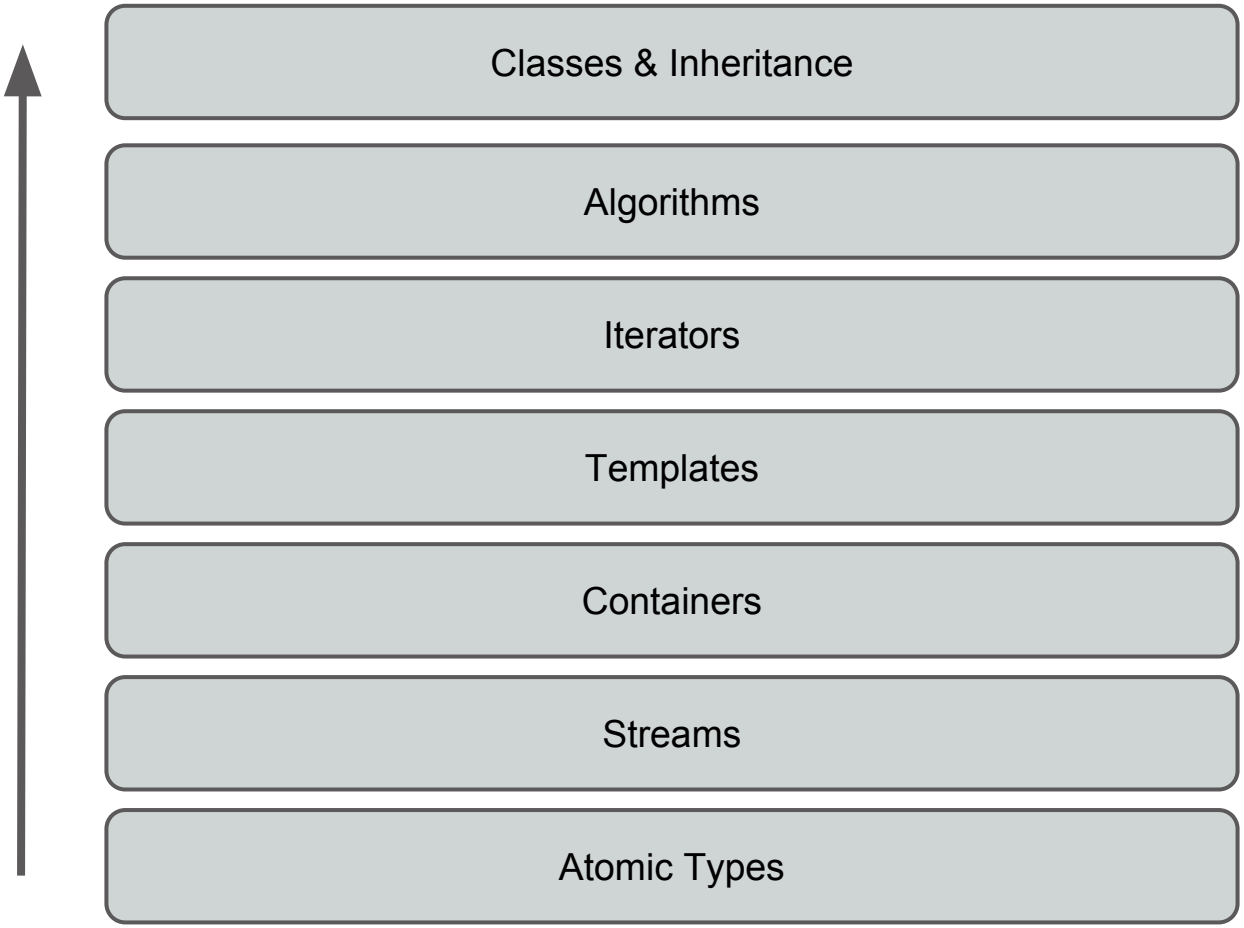
Templates & Iterators

Cristian Cibils
(ccibils@stanford.edu)

Announcements

- GraphViz due tonight at midnight
  - Well… really tomorrow night
- Assignment 2 goes out on Thursday

The Design of C++



The Path so Far

- Streams
  - More in depth than 106B/X but same concept
- Containers
  - Same as 106B/X but with some different syntax
  - Iterators to do everything
- A little bit of Iterators

So far we have been following along CS106B/X and adding in some extra details

The Path so Far

It’s time to change that!

Why Templates?

Say I want to write a function to find the minimum of two ints

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);
```

Solution 2: The C Version

In C, this problem is traditionally solved by writing different versions of the function with 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.

Solution 3: Function Overloading

In C++, you can write multiple functions with the same name as long as the have different parameters (called **Function Overloading**):

```
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 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?

Problems: This is (slightly less) 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.

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. It does not have to be called `T`

Using Templates

Using a template function is simple. Use angle brackets (<>) to indicate which type to use.

```
double x = 1.0; double y = 2.5
double smaller = min<double>(x,y);

int x = -12; int y = 42
int smaller = min<int>(x,y);
```

Using Templates

The angle brackets are actually not needed here, since the type can be **inferred**. If both the parameters are doubles, we must be using `min<double>`

```
double x = 1.0; double y = 2.5
double smaller = min(x,y);
```

Template Validation

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

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

Template Validation

Let’s take a look at what happens

when we use a method not supported by our templated type

(VectorError.pro)

Template Validation

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

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

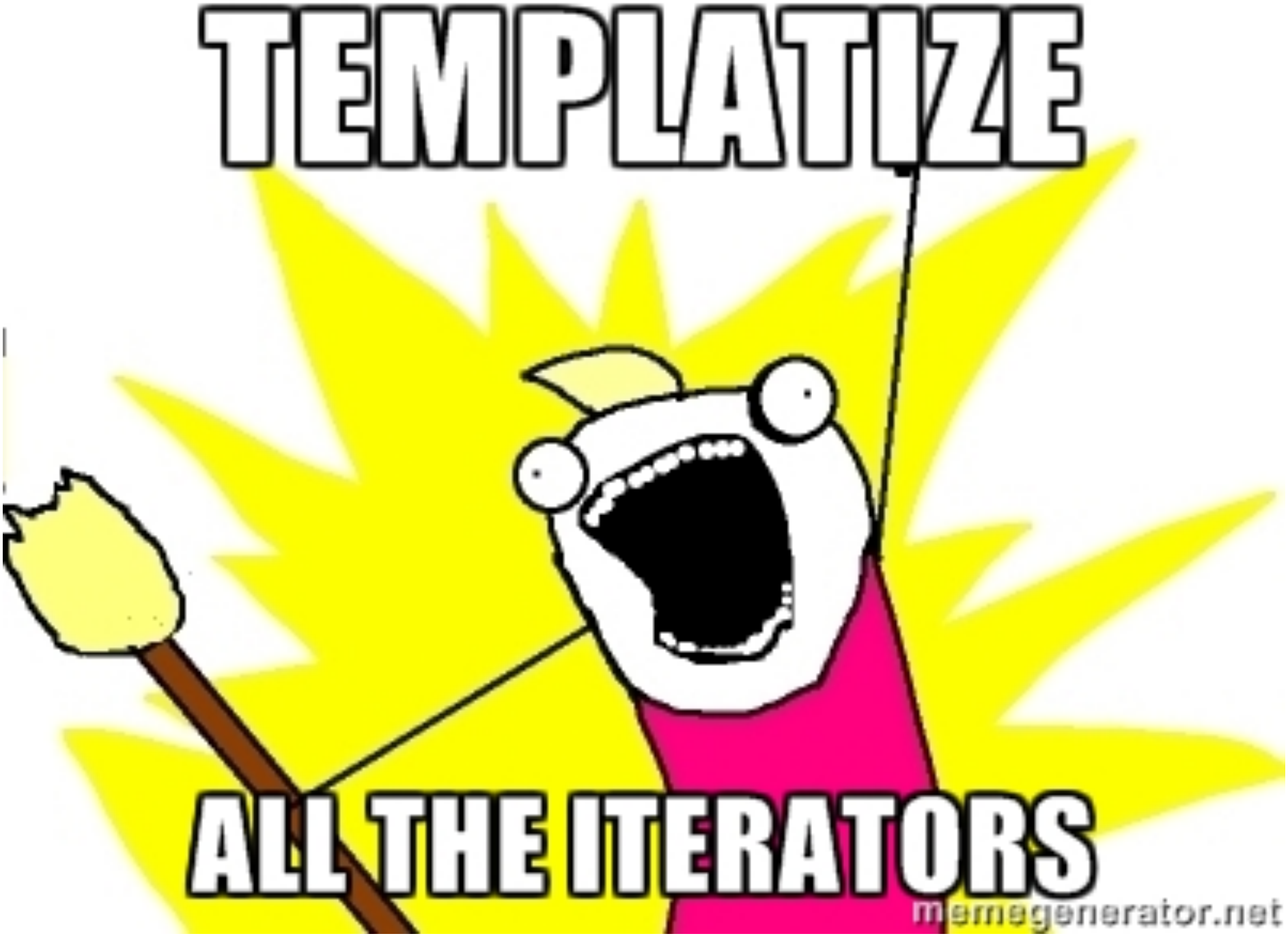
main.cpp:12:12:   required from here

main.cpp:7:5: 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

- So far we have seen templates used to lessen repetitive functions
- Templates can be used for much, much more


TEMPLATIZE
ALL THE ITERATORS
memegenerator.net

Using Templates

```
//creates an iterator for
//a vector of ints
vector<int> v;
vector<int>::iterator itr = v.begin();
```

Using Templates

```
//creates an iterator for
//a vector of ints
vector<int> v;
vector<int>::iterator itr = v.begin();
```

I don't care about what it's an iterator for when I'm writing a program -- the whole point is that I don't want to have to worry about what's behind my iterator!

Using Templates

- 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

Let’s see an example of
templates and iterators in action
(Find.pro)

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, **advanced** using ++, and **compared** to other iterators using ==:

```
vector<int> v;
v.push_back(1)
vector<int>::iterator itr = v.begin();
++itr; //itr == v.end()
if (itr == v.end()) break;
```

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 itr = v.begin();
int first = *itr; //get first element
```

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 itr = v.begin();
*itr = 2; //set first element
```

Iterator Types

Bidirectional iterators can be decremented using `--`:

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

Iterator Types

Random Access Iterators can be incremented or decremented by arbitrary amounts using `+`,`-` and related operations:

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

Using Templates and Iterators

Let's try writing code to copy the elements of one collection to another

(Copy.pro)

Using Templates and Iterators

Template Types	Parameters
typename InputIterator typename OutputIterator	InputIterator first InputIterator last OutputIterator result

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

Using Templates and Iterators

Template Types	Parameters
typename InputIterator typename OutputIterator	InputIterator first InputIterator last OutputIterator result

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

Using Templates and Iterators

Template Types	Parameters
typename InputIterator typename OutputIterator	InputIterator first InputIterator last OutputIterator result

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

Using Templates and Iterators

Let's try writing a version of copy
which will only copy certain elements
(CopyIf.pro)

Next Time

Next class we will start talking about one of the coolest parts of C++, the algorithms library

Remember to get GraphViz done!