Templates and Iterators

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Iterator Wrapup
- `auto` and Range Based `for`
- Templates
- Iterators (pt. II)
Recap
Associative Containers

Useful abstraction for “associating” a key with a value.

```cpp
std::map
    map<string, int> directory;    // name -> phone number

std::set
    set<string> dict;             // does it contains a word?
```
Iterators

Let’s try and get a mental model of iterators:

Say we have a `std::set<int> mySet`
Iterators

A standard interface to iterate through any collection.

```cpp
int numOccurences(vector<int>& cont, int elemToCount) {
  int counter = 0;
  vector<int>::iterator iter;
  for (iter = cont.begin(); iter != cont.end(); ++iter) {
    if (*iter == elemToCount)
      ++counter;
  }
  return counter;
}
```
Map Iterators

Example:

```cpp
map<int, int> m;
map<int, int>::iterator i = m.begin();
map<int, int>::iterator end = m.end();
while(i != end) {
    cout << (*i).first << (*i).second << endl;
    ++i;
}
```
Wrapping Up Iterators
auto
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:
Writing iterator types can be unsightly.

Consider a **map of deque of strings to vector of strings**:

<table>
<thead>
<tr>
<th>File Location</th>
<th>Data</th>
</tr>
</thead>
</table>
| to be | or not to be just ...                   | map = {}  
|                 |                                           | window = {to, be}  
| to be or | not to be just ...                       | map = { {to, be} : {or} }  
|                 |                                           | window = {be, or}  
| to be or not | to be just ...                            | map = { {to, be} : {or}, {be, or} : {not} }  
|                 |                                           | window = {or, not}  
| to be or not to | be just ...                               | map = { {to, be} : {or}, {be, or} : {not}, {or, not} : {to} }  
|                 |                                           | window = {not, to}  

**Where might you use this? 🤔**
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up?
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

Use an alias for the collection type!

How can we clean this up?
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NgramMap = map<deque<string>, vector<string>>;
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter =
    myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NGramMap = map<deque<string>, vector<string>>;
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up?

Use an alias for the collection type!
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NGramMap = map<deque<string>, vector<string>>;
NGramMap myMap;
for(NGramMap::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

Use an alias for the collection type!
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NGramMap = map<deque<string>, vector<string>>;
NGramMap myMap;
for (NGramMap::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up better?
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

**auto**

How can we clean this up better?

The **auto** keyword!
auto

Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}

auto keyword!
```

How can we clean this up better?
auto

auto is a C++11 feature that uses type deduction.

Asks the compiler to figure out the type for you.

When to use it?

- Use it whenever the type is obvious (e.g. iterators)
- In places where only the compiler knows the type (yes these exist)
Range Based *for* Loop

A range based *for* loop is (more or less) a shorthand for iterator code:

```cpp
map<string, int> myMap;
for(auto thing : myMap) {
    doSomething(thing.first, thing.second);
}
```

```cpp
map<string, int> myMap;
for(auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
    auto thing = *iter;
    doSomething(thing.first, thing.second);
}
```
A range based *for* loop is (more or less) a shorthand for iterator code:

```cpp
for ( auto __begin = begin-expr, __end = end-expr;
     __begin != __end;
     ++__begin ) {
  for-range-declaration = *__begin;
  statement
}
```
Templates
The Problem
Minimum Function

Let’s write a simple function to find the minimum of two ints.

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

min(3, 5); // 3
min(13, 8); // 8
min(1.9, 3.7); // 1?
```
Minimum Function

Let’s write a simple function to find the minimum of two ints.

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

- `min(3, 5);`  // 3
- `min(13, 8);`  // 8
- `min(1.9, 3.7);`  // 1?
Need more min

A classic C type solution would be to write two functions with different names:

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

double min_double(double a, double b) {
    return (a < b) ? a : b;
}
```
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_ch(char a, char b) {
    return (a < b) ? a : b;
}
And... more

```c
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_ch(char a, char b) {
    return (a < b) ? a : b;
}
```
The Problem

Multiple copies of essentially the same function.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it.

Every time you want to add a new type, you need to add a new function.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it.

Every time you want to add a new type, you need to add a new function.

If you edit the function slightly, you need to edit it in each version manually.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it

Every time you want to add a new type, you need to add a new function.

If you edit the function slightly, you need to edit it in each version manually.
Overloaded Functions

In C++, function overloading lets us have multiple functions with the same name but different parameters.

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

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

When the method is called, the compiler infers which version you mean based on the type of your parameters.
Overloaded Functions

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

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

min(3,5); // (int, int) version
min(1.9, 3.7); // (double, double) version
min(3.14, 17); // uhh.. (double, int) version?
```
Overloaded Functions

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

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

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

`min(1.9, 3.7);` // (double, double) version

`min(3.14, 17);` // uhh.. (double, int) version?
Overloaded Functions

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

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

min(3, 5); // (int, int) version
min(1.9, 3.7); // (double, double) version
min(3.14, 17); // uhh.. (double, int) version?
min(3.14, static_cast<double>(17)); // (double, double) version!
```
Overloaded Functions

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

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

min(3, 5); // (int, int) version
min(1.9, 3.7); // (double, double) version
min(3.14, 17); // uhh.. (double, int) version?
min(3.14, static_cast<double>(17)); // (double, double) version!
```

Lesson: Be explicit!
The Problems

Multiple copies of essentially the same function.

**You need to write the type name whenever you use it**

Every time you want to add a new type, you need to add a new function.

If you edit the function slightly, you need to edit it in each version manually.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it.

Every time you want to add a new type, you need to add a new function.

If you edit the function slightly, you need to edit it in each version manually.
The Problems

Multiple copies of essentially the same function.

You need to write the type name whenever you use it.

Every time you want to add a new type, you need to add a new function.

If you edit the function slightly, you need to edit it in each version manually.
What really is different in the code?
int min(int a, int b) {
    return (a < b) ? a : b;
}
double min(double a, double b) {
    return (a < b) ? a : b;
}
size_t min(size_t a, size_t b) {
    return (a < b) ? a : b;
}
float min(float a, float b) {
    return (a < b) ? a : b;
}
char min(char a, char b) {
    return (a < b) ? a : b;
}
int min(int a, int b) {
    return (a < b) ? a : b;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

char min(char a, char b) {
    return (a < b) ? a : b;
}
The Solution
Templates

Templates are a blueprint of a function that let you use the same function for a variety of types:

```cpp
template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}
```
Think about how we wrote all our min functions.

We had a set of rules (a blueprint), and the only thing we did to make different versions was to change the type.

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

Think about how we wrote all our min functions.

We had a set of rules (a blueprint), and the only thing we did to make different versions was to change the type.

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

Think about how we wrote all our min functions.

We had a set of rules (a blueprint), and the only thing we did to make different versions was to change the type.

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

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

Let’s make this general.
Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

Some generic type T.
Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

Tell compiler T is a generic type.
Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

Tell compiler T is a generic type.

It will replace the parameter for us!
Using Templates

Now we can use the template function just like any other function!

We can indicate the type through angle brackets after the function name:

```c
int a = 3, b = 9;
int c = min<int>(a, b);

double a = 3.14, b = 1.59;
double c = min<double>(a, b);
```
How does this work?

```c
int a = 3, b = 9;
int c = min<int>(a, b);
```
How does this work?

```c
int a = 3, b = 9;
int c = min<int>(a, b);
```
How does this work?

```cpp
int a = 3, b = 9;
int c = min<int>(a, b);
```

I don’t have a `min<int>` :( 
How does this work?

```c++
int a = 3, b = 9;
int c = min<int>(a, b);
```

I don’t have a `min<int>` :(

But I know how to make one!
int a = 3, b = 9;
int c = min<int>(a, b);

I don’t have a `min<int>` :(  

But I know how to make one!

template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}
How does this work?

```
int a = 3, b = 9;
int c = min<int>(a, b);
```

I don’t have a `min<int>` :(

But I know how to make one!

```
template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}
```
int a = 3, b = 9;
int c = min<int>(a, b);

How does this work?

I don’t have a min<int> :(  

But I know how to make one!

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

```cpp
int min<int>(int a, int b) {
    return (a < b) ? a : b;
}
```
You can usually omit the angle brackets when using the function.

```
int a = 3, b = 9;
int c = min<int>(a, b);
```
Templates - Aside

You can usually omit the angle brackets when using the function.

```c++
int a = 3, b = 9;
int c = min<int>(a, b);
```
Templates - Aside

You can usually omit the angle brackets when using the function.

```c
int a = 3, b = 9;
int c = min(a, b);
```
Templates - Aside

You can usually omit the angle brackets when using the function.

```c
int a = 3, b = 9;
int c = min(a, b);
```

Type inferred
Templates - Aside

You can usually omit the angle brackets when using the function.

```c
int a = 3, b = 9;
int c = min(a, b);
```

The template parameter name can be anything. I used `T` but a more descriptive name is usually better.
You can usually omit the angle brackets when using the function.

```cpp
int a = 3, b = 9;
int c = min(a, b);  // Type inferred
```

The template parameter name can be anything. I used `T` but a more descriptive name is usually better.

```cpp
template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}
```
You can usually omit the angle brackets when using the function.

```cpp
type a = 3, b = 9;
type c = min(a, b);
```

The template parameter name can be anything. I used `T` but a more descriptive name is usually better.

```cpp
template <typename Type>
    Type min(Type a, Type b) {
        return (a < b) ? a : b;
    }
```
Templates - Aside

You can usually omit the angle brackets when using the function.

```cpp
int a = 3, b = 9;
int c = min(a, b);  // Type inferred
```

The template parameter name can be anything. I used `T` but a more descriptive name is usually better.

```cpp
template <typename DataType>
DataType min(DataType a, DataType b) {
    return (a < b) ? a : b;
}
```
Questions?
Templates in Action

Let’s do a very realistic example of templates. Could be useful in assignments!

Generic Input
(GenInput.pro)
When Templates Go Wrong

Any time template instantiation occurs, the compiler will check that all operations used on the templatised type are supported by that type.

Generic Input
(GenInput.pro)

“With great power comes great responsibility”
- Uncle Ben
Errors

The start of the error is usually the most informative. For example, we got:

```
../Input/main.cpp:63:19: error: invalid operands to binary expression ('istringstream' (aka 'basic_istringstream<char>') and 'std::__1::vector<int, std::__1::allocator<int> >')
converter >> ret;
~~~~~~~~~ ^ ~~~
../Input/main.cpp:93:27: note: in instantiation of function template specialization 'getInput<std::__1::vector<int, std::__1::allocator<int> > >' requested here
vector<int> vec_inp = getInput<vector<int>>("Enter a vector (yolo): ");
```

Invalid operands to binary expression “>>” tells us we are using the stream operator on a type that doesn’t know how to work with them (in this case, vector).
Errors

The start of the error is usually the most informative. For example, we got:

```
../Input/main.cpp:63:19: error: invalid operands to binary expression ('istringstream' (aka 'basic_istringstream<char>') and 'std::__1::vector<int, std::__1::allocator<int> >')
converter >> ret;
~~~~~~~~~ ^ ~~~
../Input/main.cpp:93:27: note: in instantiation of function template specialization 'getInput<std::__1::vector<int, std::__1::allocator<int> > >' requested here
vector<int> vec_inp = getInput<vector<int>>("Enter a vector (yolo): ");
```

Invalid operands to binary expression “>>” tells us we are using the stream operator on a type that doesn’t know how to work with them (in this case, vector).
Implicit Interface

What types are valid to use with a templatized function?

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

Basically, if we replace all instances of $T$ with the actual type we want to use, would it compile?
Let’s take a moment
Templates ft. Iterators
Templates ft. Iterators

Every different collection comes equipped with its own type of iterator:

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

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

deque<int> d;
deque<int>::iterator itr = d.begin();
```
Templates ft. Iterators

The whole point of iterators was to have a standard interface to iterate over data in any container.

But we still had to specify what type of data this iterator was pointing to.

We want to ultimately write generic functions to work with iterators over any sequence.

With templates we can!
Templates ft. Iterators

With this newfound power, let’s write our first generic algorithm!

Count Occurences
(IterAlgorithms.pro)
Next Time

Algorithms