Algorithms

Ali Malik
malikali@stanford.edu
Game Plan

Recap
Iterator Types
Algorithms
Iterator Adapters
Recap
**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)
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);
}
```

How can we clean this up better?

The `auto` keyword!
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);
}
```
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;
}
```
How does this work?

```cpp
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!

```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;
}
```
Iterator Types
Iterator Types

So far we have only really incremented iterators.

But for some containers, we should be able to jump anywhere:

```cpp
std::vector<int> v(10);
auto mid = v.begin() + v.size()/2;

std::deque<int> d(13);
auto some_iter = d.begin() + 3;
```
Iterator Types

So far we have only really incremented iterators.

But for some containers, we should be able to jump anywhere:

```cpp
std::vector<int> v(10);
auto mid = v.begin() + v.size()/2;

std::deque<int> d(13);
auto some_iter = d.begin() + 3;
```
Iterator Types

But what about `std::list` (doubly linked list)?

```cpp
std::list<int> myList(10);
auto some_iter = myList.begin() + 3;
```
But what about `std::list` (doubly linked list)?

```cpp
std::list<int> myList(10);
auto some_iter = myList.begin() + 3;
```
Iterator Types

But what about `std::list` (doubly linked list)?

```cpp
std::list<int> myList(10);
auto some_iter = myList.begin() + 3;
```
Iterator Types

There are 5 different types of iterators!

1. Input
2. Output
3. Forward
4. Bidirectional
5. Random access
Iterator Types

There are 5 different types of iterators!

1. Input
2. Output
3. Forward
4. Bidirectional
5. Random access
Iterator Types - Similarities

All iterators share a few common traits:

- Can be created from existing iterator
- Can be advanced using `++`
- Can be compared with `==` and `!=`
Input Iterators

For sequential, single-pass input.

Read only i.e. can only be dereferenced on right side of expression.

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

For sequential, single-pass output.

Write only i.e. can only be dereferenced on left side of expression.

```cpp
vector<int> v = ...;  
vector<int>::iterator itr = v.begin();  
*itr = 12;
```
Forward Iterators

Same as input/output iterators except can make multiple passes.

Can read from and write to (if not const iterator).

```cpp
vector<int> v = ...;
vector<int>::iterator itr = v.begin();
int val = *itr;
int val2 = *itr;
```
Bidirectional Iterators

Same as forward iterators except can also go backwards with the decrement operator `--`.

```cpp
vector<int> v = ...;
vector<int>::iterator itr = v.begin();
int val = *itr;
--itr;
int val2 = *itr;
```
Random Access Iterators

Same as bidirectional iterators except can be incremented or decremented by arbitrary amounts using + and –.

```cpp
vector<int> v = ...;
vector<int>::iterator itr = v.begin();
int val = *itr;
itr = itr + 3;
int val2 = *itr;
```
Algorithms
Abstraction in the STL

Abstractions allow us to express the general structure of a problem instead of the particulars of its implementation.

If we solve problems in a general setting, we solve all specific instances of the problem!
Abstraction in the STL

We began by talking about *basic* types:

- `char`
- `int`
- `double`
- `...`

Each type was conceptually a “*single value*”.
Abstraction in the STL

Basic Types
Abstraction in the STL

Many programs require a collection of basic types:

- A `vector<int>` representing student ages
- A `map<string, int>` of names to phone numbers

Containers allow a programmer to use the same collection, regardless of the underlying type.
Abstraction in the STL

Containers

Basic Types
Abstraction in the STL

The same `<vector>` implementation can be used for any basic type.

Containers let us perform operations on basic types, regardless of what the basic type is.

Can we perform operations on containers regardless of what the container is?
Abstraction in the STL

Iterators allow us to abstract away from the container being used.

Similar to how containers allow us to abstract away from the basic type being used.
Abstraction in the STL

- Iterators
- Containers
- Basic Types
Abstraction in the STL

Operations like sorting, searching, filtering, partitioning etc. can be written to work with almost any container.

If we write algorithms that operate on iterators, then they will be applicable in the most general setting.
Abstraction in the STL

- Basic Types
- Containers
- Iterators
- Algorithms
Algorithms

The STL contains pre written algorithms that operate on iterators.

Doing so lets them work on many types of containers.

Uses determined by types of iterators.

Rely heavily on templates.
Algorithms

Let’s have some fun with algorithms:

Algorithm Fun
(AlgorithmFun.pro)
In depth - `std::copy`

Let’s look at the `std::copy` algorithm to get a better understanding of algorithms and iterators:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(v.size());

std::copy(v.begin(), v.end(), vCopy.begin());
```
In depth - `std::copy`

v:

| 561 | 1105 | 1729 | 2465 |

vcopy:

| 0   | 0    | 0    | 0    | 0    |
In depth - `std::copy`

v:

| 561 | 1105 | 1729 | 2465 |

vcopy:

| 561 | 0    | 0    | 0    | 0    |
In depth - `std::copy`

<table>
<thead>
<tr>
<th>v:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vcopy:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
In depth - `std::copy`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

v:

vcopy:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
</tr>
</tbody>
</table>

0
In depth - `std::copy`

<table>
<thead>
<tr>
<th>561</th>
<th>1105</th>
<th>1729</th>
<th>2465</th>
</tr>
</thead>
</table>

`v:

561 1105 1729 2465`

`vcopy:

561 1105 1729 2465`
In depth - `std::copy`

What happens if there isn’t enough space in the destination?

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(v.size());
std::copy(v.begin(), v.end(), vCopy.begin());
```
In depth - `std::copy`

What happens if there isn’t enough space in the destination?

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(v.size());

std::copy(v.begin(), v.end(), vCopy.begin());
```
In depth - `std::copy`

What happens if there isn’t enough space in the destination?

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(2);

std::copy(v.begin(), v.end(), vCopy.begin());
```
In depth - std::copy

v:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

vcopy:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
In depth - `std::copy`

v:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

vcopy:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>0</td>
</tr>
</tbody>
</table>
In depth - std::copy

\[\text{v:} \]

\[
\begin{array}{cccc}
561 & 1105 & 1729 & 2465 \\
\end{array}
\]

\[\text{vcopy:} \]

\[
\begin{array}{cc}
561 & 1105 \\
\end{array}
\]
In depth - `std::copy`

v:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

vcopy:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
</tr>
</tbody>
</table>
**In depth - std::copy**

<table>
<thead>
<tr>
<th>v:</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vcopy:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td></td>
</tr>
</tbody>
</table>
In depth - `std::copy`

We won’t always know how much space will be needed for the destination?

How can we solve this problem?

We want to be able to copy into a collection by “inserting” into it, rather than making space for it first.

C++ has a solution!
Iterator Adapters
This is where iterator adaptors come in.

Iterator Adaptors

Sometimes we need to form “weird” iterators:

- Iterating over streams would be pretty cool
- Having an iterator that could “insert” into a collection would be pretty cool
Iterator Adaptors

Act like iterators:

- Can be dereferenced with \(*\)
- Can be advanced with \(++\)

However, they don’t actually point to elements of a container.
`std::ostream_iterator`

Look like output iterators

- Can be dereferenced with `*`
- Can be advanced with `++`

Whenever you dereference a `std::ostream_iterator` and assign a value to it, the value is printed to a specified `std::ostream`. 
std::ostream_iterator

Example:

```cpp
std::ostream_iterator<int> itr(cout, "", "");
*itr = 3; // prints 3 to console
++itr;
*itr = 1729; // prints 1729 to console
++itr;
*itr = 13; // prints 13 to console
```

Output: 3, 1729, 13,
std::ostream_iterator

Looks like you’re manipulating contents of a container.

... 

But really you’re writing characters to the cout stream.
`std::ostream_iterator` looks like you’re manipulating contents of a container.

... But really you’re writing characters to the cout stream.
std::ostream_iterator

What is this even useful for?

You can treat streams like iterators, so you can use algorithms with them!
Here’s a cool application of this. This code prints the vector:

```cpp
std::vector<int> v{3, 1, 4, 1, 5};
std::copy(v.begin(), v.end(),
          std::ostream_iterator<int>(cout, " "))
```
Iterator Adapters
Insert Iterators
std::back_inserter

Let’s get back to the original problem.
In depth - `std::copy`

| v:         |
|---|---|---|---|
| 561 | 1105 | 1729 | 2465 |

| vcopy:     |
|---|---|
| 561 | 1105 |
Insert Iterators

Let’s get back to the original problem.

We want to be able to copy into a collection by “inserting” into it, rather than making space for it first.
Insert Iterators

The standard library provides insert iterators (`std::inserter`, `std::back_inserter`, `std::front_inserter`).

Writing to these iterators inserts the value into a container using one of `insert`, `push_back`, or `push_front`. 
Insert Iterators

Example:

```cpp
std::vector<int> v;   // empty vec
auto itr = std::back_inserter(v);
*itr = 1729;          // does v.push_back(1729)
++itr;
*itr = 13;            // does v.push_back(13)
++itr;
*itr = 3;             // does v.push_back(3)
```

v look like this: {1729, 13, 3}
Insert Iterators

Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(v.size());
std::copy(v.begin(), v.end(), vCopy.begin());
```
Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy(v.size());
std::copy(v.begin(), v.end(), vCopy.begin());
```
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy;

std::copy(v.begin(), v.end(), vCopy.begin());
Insert Iterators

Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy;
std::copy(v.begin(), v.end(), vCopy.begin());
```
Insert Iterators

Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy;

std::copy(v.begin(), v.end(), vCopy.begin());
```
Insert Iterators

Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy;

std::copy(v.begin(), v.end(), vCopy.begin());
```
Insert Iterators

Now we can solve the coyote problem:

```cpp
vector<int> v {561, 1105, 1729, 2465};
vector<int> vCopy;
std::copy(v.begin(), v.end(), std::back_inserter(vCopy));
```
In depth - `std::copy`

v:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>561</td>
<td>1105</td>
<td>1729</td>
<td>2465</td>
</tr>
</tbody>
</table>

vcopy:
In depth - std::copy

| 561 | 1105 | 1729 | 2465 |

v:

vcopy:

```cpp
561
```
In depth - `std::copy`

```cpp
v:
```

<table>
<thead>
<tr>
<th></th>
<th>561</th>
<th>1105</th>
<th>1729</th>
<th>2465</th>
</tr>
</thead>
</table>
```

vcopy:

<table>
<thead>
<tr>
<th></th>
<th>561</th>
<th>1105</th>
</tr>
</thead>
</table>
In depth - `std::copy`

v:

| 561 | 1105 | 1729 | 2465 |

vcopy:

| 561 | 1105 | 1729 |
In depth - `std::copy`

v:

| 561 | 1105 | 1729 | 2465 |

vcopy:

| 561 | 1105 | 1729 | 2465 |
Closing Notes

There are many algorithms we didn’t cover today. [Here](#) is a full list of them.

We will be using algorithms heavily for the next assignment.

The course reader does a really good job on this topic, so please check it out!