Associative Containers and Iterators

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Associative Containers
- Iterators
- Map Iterators
- The `auto` keyword (maybe)
- Range-Based `for` Loops (maybe)
Recap
You can define your own mini-types that bundle multiple variables together:

```c
struct point {
    int x;
    int y;
};
```

Useful for Assignment 1
Overview of STL
Sequence Containers

Provides access to sequences of elements.

Examples:

- `std::vector<T>`
- `std::list<T>`
- `std::deque<T>`
Some new stuff there:

```cpp
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```

This let’s us use `vecsz_t` as an alias/synonym for the type `std::vector<int>::size_type`;
std::deque<T>

A deque (pronounced “deck”) is a double ended queue. Can do everything a vector can do

and also...

Unlike a vector, it is possible (and fast) to push_front and pop_front.
Container Adapters

How can we implement stack and queue using the containers we have?

Stack:
Just limit the functionality of a vector/deque to only allow `push_back` and `pop_back`.

Queue:
Just limit the functionality of a deque to only allow `push_back` and `pop_front`.

Plus only allow access to top element
Associative Containers
Scenario:
You want to count the frequency of words in a file.

What do you use?
- vector<string>, vector<int>
Scenario:
You want to count the frequency of words in a file.

What do you use?
- `vector<string>; vector<int>`

We would need to keep two vectors with indexes denoting pairs.
Scenario:
You want to count the frequency of words in a file.

What do you use?

- vector<string>; vector<int>
**Scenario:**

You want to count the frequency of words in a file.

What do you use?

- `vector<string>; vector<int>`
- `vector<std::pair<string, int>>`
Scenario:
You want to count the frequency of words in a file.

What do you use?
- `vector<string>; vector<int>`
- `vector<std::pair<string, int>>`

No quick way to lookup based on word.
Scenario:
You want to count the frequency of words in a file.

What do you use?
- `vector<string>; vector<int>`
- `vector<std::pair<string, int>>`
Associative Container

Scenario:
You want to count the frequency of words in a file.

What do you use?
- `vector<string>, vector<int>`
- `vector<tuple<string, int>>`
- `std::map<string, int>`
Associative Container

Scenario:
You want to count the frequency of words in a file.

What do you use?
- `vector<string>; vector<int>`
- `vector<std::pair<string,int>>`
- `std::map<string, int>`
It is useful to have a data structure that can associate values.

Associative containers do exactly that!
Have no idea of a sequence.

Data is accessed using the key instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Associative Container

Have no idea of a sequence.

Data is accessed using the **key** instead of **indexes**.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Have no idea of a sequence.

Data is accessed using the key instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

Based on ordering property of keys.

Keys need to be comparable using `<` (less than) operator.
Associate Container

Have no idea of a sequence.

Data is accessed using the **key** instead of **indexes**.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Have no idea of a sequence.

Data is accessed using the key instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Associative Container

Have no idea of a sequence.

Data is accessed using the key instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

Based on hash function. You need to define how the key can be hashed.
Have no idea of a sequence.

Data is accessed using the key instead of indexes.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Have no idea of a sequence.

Data is accessed using the **key** instead of **indexes**.

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

You can define `<` and hash function operators for your own classes!
Methods mostly same as Stanford map.

Check documentation for full list of methods.

We can do a small example, counting frequency of words in a file:

\texttt{Map Example (MapExample.pro)}
std::set<T>

Methods mostly same as Stanford map.

Check [documentation](#) for full list of methods.

Key point:
A set is just a specific case of a map that doesn’t have a value.

We can do a small example of adding and removing stuff:

Set Example
:SetExample.pro
Announcements
Game Plan

- Recap
- Associative Containers
- Iterators
- Map Iterator
- The `auto` keyword
- Range-Based `for` Loops
How do we iterate over associative containers?

Remember:
Assoc. containers have no notion of a sequence/indexing

```cpp
for(int i = umm?; i < uhh?; i++ maybe?) {
```
Iterators

How do we iterate over associative containers?

Remember:
Assoc. containers have no notion of a sequence/indexing

```cpp
for(int i = umm?; i < uhh?; i++ maybe?) {
```
Iterators

How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing

for(int i = umm?; i < uhh?; i++ maybe?) {

C++ has a solution!
Iterators
First: A note
A note

We are going on a journey.
We could in theory fly to the end destination.
A note

But then we would miss the experience
A note

We are going on a journey.
A note

We are going on a journey.

At the end lies simplicity, but if we jump to it we miss out on understanding.
A note

We are going on a journey.

At the end lies simplicity, but if we jump to it we miss out on understanding.

So we will walk to it.
Iterators
Iterators

Iterators allow iteration over any container, whether it is ordered or not.
Iterators

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

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

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

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

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

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

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

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

Iterators let us view a non-linear collection in a linear manner.
Iterators

How do they work?

We don’t care right now.

We will just use them like any other thing - assume they just work somehow.
Iterators - Usage

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

We can get an iterator pointing to the “start” of the sequence by calling `mySet.begin()`.

```
mySet.begin();
```
Iterators - Usage

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

mySet.begin();
Let’s try and get a mental model of iterators:

```cpp
mySet.begin();
```
Iterators - Usage

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

```c++
??? iter = mySet.begin();
```
Iterators - Usage

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

```cpp
??? iter = mySet.begin();
```

What is the type of the iterator?
Iterators - Usage

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

```cpp
set<int> mySet;
mySet.begin();
```

What is the type of the iterator?

```cpp
??? iter = mySet.begin();
```
Iterators - Usage

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

```c++
???
iter = mySet.begin();
```
Let's try and get a mental model of iterators:

```
set<int>::iterator iter = mySet.begin();
```
Iterators - Usage

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

```
set<int>::iterator iter = mySet.begin();
```

It is the iterator type defined in the `set<int>` class!
Let’s try and get a mental model of iterators:

```
set<int>::iterator iter = mySet.begin();
```
Iterators - Usage

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

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

We can get the value of an iterator by using the dereference * operator.
Iterators - Usage

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

We can get the value of an iterator by using the dereference * operator.

cout << *iter << endl; // prints 1
Iterators - Usage

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

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

We can advance the iterator one by using the \texttt{++} operator (prefix)
Iterators - Usage

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

We can advance the iterator one by using the ++ operator (prefix)

```
++iter;  // advances iterator
```
Iterators - Usage

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

And so on...
Iterators - Usage

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

```
1 2 3 4
```

```
cout << *iter << endl;  // prints 2
```
Iterators - Usage

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

And so on...
Iterators - Usage

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

```
1 2 3 4
```

```
++iter; // advances iterator
```

And so on...
Iterators - Usage

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

```
1 2 3 4
```

And so on...
Iterators - Usage

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

```
cout << *iter << endl;  // prints 3
```
Iterators - Usage

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

1  2  3  4

And so on...
Iterators - Usage

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

```
++iter;       // advances iterator
```

And so on...
Iterators - Usage

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

```
1 2 3 4
iter
```

And so on...
Let’s try and get a mental model of iterators:

```cpp
cout << *iter << endl;  // prints 4
```
Iterators - Usage

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

```
1 2 3 4
```

And so on...
Iterators - Usage

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

++iter;  // advances iterator
Iterators - Usage

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

```plaintext
1 2 3 4
```

iter
Iterators - Usage

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

We can check if we have hit the end by comparing to `mySet.end()`.
Iterators - Usage

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

We can check if we have hit the end by comparing to `mySet.end()`

```cpp
if(iter == mySet.end()) return;
```
Iterators - Usage

A summary of the essential iterator operations:

- **Create** iterator
- **Dereference** iterator to read value currently pointed to
- **Advance** iterator
- **Compare** against another iterator (especially `.end()` iterator)
Iterators - Usage

Let’s do some examples:

Basic Iterator
(BasicIter.pro)
Iterators

Our examples have used sets, but (almost) all C++ containers have iterators.

Why is this powerful?

- Many scenarios require looking at elements, regardless of what type of container is storing those elements.
- Iterators let us go through sequences of elements in a standardised way.
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(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;
}
```
Iterators

Example (find number occurrences):

```c++
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;
}
```

Can I make this work for `std::list<int>`?
Iterators

Example (find number occurrences):

```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;
}
```

Can I make this work for std::list<int>?
Iterators

Example (find number occurrences):

```cpp
int numOccurences(list<int>& cont, int elemToCount) {
    int counter = 0;
    list<int>::iterator iter;
    for(iter = cont.begin(); iter != cont.end(); ++iter) {
        if(*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```
This standard interface for looping through things is going to be really powerful.

We will cover it sometime this week or next week!
Map Iterators
Map Iterators

Map iterators are slightly different because we have both keys and values.

The iterator of a `map<string, int>` points to a `std::pair<string, int>`.
The **std::pair** Class

A pair is simply two objects bundled together.

Syntax:

```cpp
std::pair<string, int> p;
p.first = "Phone number";
p.second = 6504550404;
```
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;
}
```
Next Time

Templates and Iterators