Agenda

Today we will discuss:

1. Announcements
2. Concepts, Conceptually
3. Iterators
   - Definition
   - The Iterator Concepts
     - Trivial Iterator
     - Input Iterator
     - Output Iterator
     - Forward Iterator
     - Bidirectional Iterator
     - Random Access Iterator
   - Iterators over STL Containers
4. Next Time
1 Announcements
2 Concepts, Conceptually
3 Iterators
   • Definition
   • The Iterator Concepts
     • Trivial Iterator
     • Input Iterator
     • Output Iterator
     • Forward Iterator
     • Bidirectional Iterator
   • Iterators over STL Containers
4 Next Time
Announcements

- MP0 is due Wednesday (02/13) at 11:59pm on Bitbucket.
Announcements

- MP0 is due Wednesday (02/13) at 11:59pm on Bitbucket.
- If you haven’t contacted me for read access to the repository... shame on you.
Announcements

- MP0 is due Wednesday (02/13) at 11:59pm on Bitbucket.
- If you haven’t contacted me for read access to the repository... shame on you.
- MP1 is out! Due: Wednesday, February 27th at 11:59pm on Bitbucket.
Announcements

- MP0 is due Wednesday (02/13) at 11:59pm on Bitbucket.
- If you haven't contacted me for read access to the repository... shame on you.
- MP1 is out! Due: Wednesday, February 27th at 11:59pm on Bitbucket.
- Assignment page: https://chara.cs.illinois.edu/cs296/assignment/mp1.html
Announcements

- MP0 is due Wednesday (02/13) at 11:59pm on Bitbucket.
- If you haven’t contacted me for read access to the repository... shame on you.
- MP1 is out! Due: Wednesday, February 27th at 11:59pm on Bitbucket.
- Assignment page: https://chara.cs.illinois.edu/cs296/assignment/mp1.html
- Two week deadline—this is a longer assignment (and there’s a heap of provided code)
Announcements

Concepts, Conceptually

Iterators
- Definition
- The Iterator Concepts
  - Trivial Iterator
  - Input Iterator
  - Output Iterator
  - Forward Iterator
  - Bidirectional Iterator
- Random Access Iterator
- Iterators over STL Containers

Next Time
A **concept** in C++ is a mechanism by which the behavior of a template type is specified.
A **concept** in C++ is a mechanism by which the behavior of a template type is specified.

- Think: interface, but without an imposed inheritance hierarchy/vtable lookups
Concepts

Definition (Concept)

A **concept** in C++ is a mechanism by which the behavior of a template type is specified.

- Think: interface, but without an imposed inheritance hierarchy/vtable lookups
- At present, they do not exist formally in the language specification.
Definition (Concept)

A concept in C++ is a mechanism by which the behavior of a template type is specified.

- Think: interface, but without an imposed inheritance hierarchy/vtable lookups
- At present, they do not exist formally in the language specification.
- An extension to C++’s template system was proposed for C++11
A concept in C++ is a mechanism by which the behavior of a template type is specified.

- Think: interface, but without an imposed inheritance hierarchy/vtable lookups
- At present, they do not exist formally in the language specification.
- An extension to C++’s template system was proposed for C++11
- ...but later removed by the committee (viewed as not ready, too complicated)
Concepts—Who Cares? (You do)

- Despite being voted out of inclusion in C++11, they still exist in documentation about the language
Concepts—Who Cares? (You do)

- Despite being voted out of inclusion in C++11, they still exist in documentation about the language.
- Much of the behavior of classes in the STL is dictated by a set of Concepts that the classes conform to.
Despite being voted out of inclusion in C++11, they still exist in documentation about the language.

Much of the behavior of classes in the STL is dictated by a set of Concepts that the classes conform to.

Documentation for generic algorithms in the STL often mention what concepts template parameters must conform to.
Concepts—Who Cares? (You do)

- Despite being voted out of inclusion in C++11, they still exist in documentation about the language.
- Much of the behavior of classes in the STL is dictated by a set of Concepts that the classes conform to.
- Documentation for generic algorithms in the STL often mention what concepts template parameters must conform to.
- Can be arranged into hierarchies, where concepts refine one another.
Despite being voted out of inclusion in C++11, they still exist in documentation about the language.

Much of the behavior of classes in the STL is dictated by a set of Concepts that the classes conform to.

Documentation for generic algorithms in the STL often mention what concepts template parameters must conform to.

Can be arranged into hierarchies, where concepts refine one another.

You will investigate an important hierarchy of concepts in MP1: the Iterators.
Announcements

Concepts, Conceptually

Iterators
  • Definition
  • The Iterator Concepts
    • Trivial Iterator
    • Input Iterator
    • Output Iterator
    • Forward Iterator
    • Bidirectional Iterator
    • Random Access Iterator
  • Iterators over STL Containers

Next Time
Definition (Iterator)

An **iterator** is a generalization of a pointer: it is an object that points to another object. It is (typically) used for iterating over a sequence of such objects.

- A pointer to an element in an array is a basic example of an iterator
Definition (Iterator)

An **iterator** is a generalization of a pointer: it is an object that points to another object. It is (typically) used for iterating over a sequence of such objects.

- A pointer to an element in an array is a basic example of an iterator
- There are certain concepts that a pointer to an element in an array adheres to, as well
An **iterator** is a generalization of a pointer: it is an object that points to another object. It is (typically) used for iterating over a sequence of such objects.

- A pointer to an element in an array is a basic example of an iterator.
- There are certain concepts that a pointer to an element in an array adheres to, as well.
- The STL defines **six** different iterator concepts (credit for much of the proceeding slides goes to the following link): [http://www.sgi.com/tech/stl/Iterators.html](http://www.sgi.com/tech/stl/Iterators.html)
Concept (Trivial Iterator)

A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.
A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*. 

Example:

```cpp
int * p = new int(5);
p is a Trivial Iterator.
```
Concept (Trivial Iterator)

A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*.
- **Mutable** implies that the pointee is assignable through the iterator.
A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*.
- **Mutable** implies that the pointee is assignable through the iterator
- **Immutable** implies that the pointee is *not* assignable through the iterator
Trivial Iterator

Concept (Trivial Iterator)

A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*.
- **Mutable** implies that the pointee is assignable through the iterator
- **Immutable** implies that the pointee is *not* assignable through the iterator
- The object obtained via dereference is called the value_type.
Trivial Iterator

Concept (Trivial Iterator)

A **Trivial Iterator** is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*.
- **Mutable** implies that the pointee is assignable through the iterator
- **Immutable** implies that the pointee is *not* assignable through the iterator
- The object obtained via dereference is called the `value_type`.
- Example:
  ```c++
  int * p = new int(5);
  ```
Trivial Iterator

Concept (Trivial Iterator)

A Trivial Iterator is an object which can be dereferenced to obtain another object. It does not have to support increment or comparison operators—it simply can be dereferenced.

- Can either be *mutable* or *immutable*.
- **Mutable** implies that the pointee is assignable through the iterator
- **Immutable** implies that the pointee is *not* assignable through the iterator
- The object obtained via dereference is called the *value_type*.
- Example:
  ```cpp
  int * p = new int(5);
  ```
- p is a Trivial Iterator.
Concept (Input Iterator)

An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.
Input Iterator

Concept (Input Iterator)

An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a refinement of Trivial Iterator.
An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a **refinement** of Trivial Iterator.
- They can be compared with operator\(==\).
An Input Iterator is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a refinement of Trivial Iterator.
- They can be compared with operator==.
- Distance between two can be measured, and will fit in the iterator’s distance_type.
Concept (Input Iterator)

An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a **refinement** of Trivial Iterator.
- They can be compared with `operator==`.
- Distance between two can be measured, and will fit in the iterator’s `distance_type`.
- Once you increment one, you aren’t guaranteed that a copy of the old one is still valid.
Concept (Input Iterator)

An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a **refinement** of Trivial Iterator.
- They can be compared with `operator==`.
- Distance between two can be measured, and will fit in the iterator’s `distance_type`.
- Once you increment one, you aren’t guaranteed that a copy of the old one is still valid.
- Example:
  ```cpp
  std::istream_iterator<int> in( std::cin );
  ```
Input Iterator

Concept (Input Iterator)

An **Input Iterator** is an object which can be dereferenced to obtain another object, and can be incremented to obtain an iterator to the next element in the sequence.

- This is a **refinement** of Trivial Iterator.
- They can be compared with `operator==`.
- Distance between two can be measured, and will fit in the iterator’s `distance_type`.
- Once you increment one, you aren’t guaranteed that a copy of the old one is still valid.
- Example:

  ```cpp
  std::istream_iterator<int> in( std::cin );
  ```
  
  `in` is an **Input Iterator**.
Output Iterator

Concept (Output Iterator)

An **Output Iterator** is an iterator which may be used to store a sequence of values.
Concept (Output Iterator)

An **Output Iterator** is an iterator which may be used to store a sequence of values.

- Refinement of Trivial Iterator.
Output Iterator

Concept (Output Iterator)

An **Output Iterator** is an iterator which may be used to store a sequence of values.

- Refinement of Trivial Iterator.
- Analogy: a tape. Opposite of **Input Iterator**.
Concept (Output Iterator)

An **Output Iterator** is an iterator which may be used to store a sequence of values.

- Refinement of Trivial Iterator.
- Analogy: a tape. Opposite of **Input Iterator**.
- Is not obligated implement `operator==`. 

Example:

```cpp
    std::vector<int> myvec;
    std::back_insert_iterator< std::vector<int> > it( myvec );
    *it = 5;
    it  // is an Output Iterator.
```
An **Output Iterator** is an iterator which may be used to store a sequence of values.

- Refinement of Trivial Iterator.
- Analogy: a tape. Opposite of **Input Iterator**.
- Is not obligated implement `operator==`.
- Example:

```cpp
std::vector<int> myvec;
std::back_insert_iterator< std::vector<int> > it( myvec );
*it = 5;
```
Output Iterator

Concept (Output Iterator)

An **Output Iterator** is an iterator which may be used to store a sequence of values.

- Refinement of Trivial Iterator.
- Analogy: a tape. Opposite of **Input Iterator**.
- Is not obligated implement `operator==`.
- Example:
  ```cpp
  std::vector<int> myvec;
  std::back_insert_iterator< std::vector<int> > it( myvec );
  *it = 5;
  ```
- **it** is an **Output Iterator**.
A **Forward Iterator** is an iterator which can be dereferenced and incremented.
A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
- Can be either *mutable* or *immutable*.
A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
- Can be either *mutable* or *immutable*.
- Once you increment one, a copy of the old one is still valid, unlike Input Iterator.
A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
- Can be either *mutable* or *immutable*.
- Once you increment one, a copy of the old one is still valid, unlike Input Iterator.
- Example:
  ```cpp
  std::unordered_set<int>::iterator it;
  ```
Concept (Forward Iterator)

A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
- Can be either *mutable* or *immutable*.
- Once you increment one, a copy of the old one is still valid, unlike Input Iterator.
- Example:
  ```cpp
  std::unordered_set<int>::iterator it;
  it is a **Forward Iterator**.
  ```
A **Forward Iterator** is an iterator which can be dereferenced and incremented.

- Refinement of Input Iterator and/or Output Iterator.
- Can be either *mutable* or *immutable*.
- Once you increment one, a copy of the old one is still valid, unlike Input Iterator.

**Example:**

```cpp
std::unordered_set<int>::iterator it;
```

- it is a **Forward Iterator**.
- (we’ll talk about unordered_set later...)
A Bidirectional Iterator is an iterator which can be dereferenced, incremented, and decremented.
Bidirectional Iterator

Concept (Bidirectional Iterator)

A **Bidirectional Iterator** is an iterator which can be dereferenced, incremented, and decremented.

- A refinement of Forward Iterator.
Bidirectional Iterator

Concept (Bidirectional Iterator)

A **Bidirectional Iterator** is an iterator which can be dereferenced, incremented, and decremented.

- A refinement of Forward Iterator.
- Can be *mutable* or *immutable*.
A **Bidirectional Iterator** is an iterator which can be dereferenced, incremented, and decremented.

- A refinement of Forward Iterator.
- Can be *mutable* or *immutable*.
- Once you increment or decrement one, a copy of an older one is still valid.
A **Bidirectional Iterator** is an iterator which can be dereferenced, incremented, and decremented.

- A refinement of Forward Iterator.
- Can be *mutable* or *immutable*.
- Once you increment or decrement one, a copy of an older one is still valid.
- Example:
  ```cpp
  std::list<int>::iterator it;
  ```
A **Bidirectional Iterator** is an iterator which can be dereferenced, incremented, and decremented.

- A refinement of Forward Iterator.
- Can be *mutable* or *immutable*.
- Once you increment or decrement one, a copy of an older one is still valid.
- Example:
  ```cpp
  std::list<int>::iterator it;
  ```
- it is a **Bidirectional Iterator**.
Random Access Iterator

Concept (Random Access Iterator)

A **Random Access Iterator** is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.
Concept (Random Access Iterator)

A **Random Access Iterator** is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
Random Access Iterator

Concept (Random Access Iterator)

A **Random Access Iterator** is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
- Supports essentially all operations of C pointer arithmetic.
A **Random Access Iterator** is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
- Supports essentially all operations of C pointer arithmetic.
- Supports `operator<`, which provides a strict weak ordering of the iterators.
Random Access Iterator

Concept (Random Access Iterator)

A Random Access Iterator is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
- Supports essentially all operations of C pointer arithmetic.
- Supports operator<, which provides a strict weak ordering of the iterators.
  - \( a < b \) if either \( b \) is reachable from \( a \), or vice versa, or both.
A Random Access Iterator is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
- Supports essentially all operations of C pointer arithmetic.
- Supports operator<, which provides a strict weak ordering of the iterators.
  - a < b if either b is reachable from a, or vice versa, or both

Example:
```cpp
std::vector<int>::iterator it;
```
Random Access Iterator

Concept (Random Access Iterator)

A Random Access Iterator is an iterator which can be dereferenced, but also incremented and decremented in arbitrarily sized steps.

- A refinement of Bidirectional Iterator and LessThan Comparable.
- Supports essentially all operations of C pointer arithmetic.
- Supports operator<, which provides a strict weak ordering of the iterators.
  - \( a < b \) if either \( b \) is reachable from \( a \), or vice versa, or both

Example:

```cpp
std::vector<int>::iterator it;
```

- it is a Random Access Iterator.
The STL provides many different container classes
The STL provides many different container classes

Typically (but not always), these classes provide mechanisms for creating iterators (or `const_iterators`) over elements in the collection.
Containers in the STL

- The STL provides many different container classes
- Typically (but not always), these classes provide mechanisms for creating iterators (or const_iterators) over elements in the collection
- Each container’s iterator may conform to a different iterator concept, which limits the things that may be done with iterators from that class
Using Iterators with STL Containers

```cpp
std::vector<int> myvec;
/*
 * some initialization code here...
*/
typename std::vector<int>::iterator it;
for( it = myvec.begin(); it != myvec.end(); ++it )
    // do something cool

// Why typename?
```
std::vector<int> myvec;

/*
 * some initialization code here...
 */

typename std::vector<int>::iterator it;
for( it = myvec.begin(); it != myvec.end(); ++it )
    // do something cool

// Why typename?
Using Iterators with STL Containers

```cpp
std::vector<int> myvec;

/*
 * some initialization code here...
*/

typename std::vector<int>::iterator it;
for( it = myvec.begin(); it != myvec.end(); ++it )
    // do something cool

// Why typename?

- Containers will almost universally specify a `begin()` and `end()` function to get iterators to the beginning and end of the collection, respectively.
```
Announcements

Concepts, Conceptually

Iterators
- Definition
- The Iterator Concepts
  - Trivial Iterator
  - Input Iterator
  - Output Iterator
  - Forward Iterator
  - Bidirectional Iterator
  - Random Access Iterator
- Iterators over STL Containers

Next Time
Next lecture:

- Containers and Algorithms
Next lecture:

- Containers and Algorithms
- Generic Programming in the New Standard (brief)
Next lecture:

- Containers and Algorithms
- Generic Programming in the New Standard (brief)

Reminder: MP0 is due **this Wednesday, February 13th at 11:59pm**! Don’t wait until the last minute—the report will take some time.
Next lecture:

- Containers and Algorithms
- Generic Programming in the New Standard (brief)

Reminder: MP0 is due **this Wednesday, February 13th at 11:59pm**! Don’t wait until the last minute—the report will take some time.

Start looking at the code for MP1, and start when you feel comfortable!