Chapter 20: Containers and Iterators

Plan for today

- We will talk about:
  - STL
  - containers
  - sequences and iterators
  - The simplest algorithm: find()
  - Parameterizaion of algorithms
    - find_if() and function objects

STL

- C++ standard library provides a framework for dealing with data as sequences of elements, called STL (Standard Template Library).
- STL provides containers (`vector`, `list`, `map`, etc) and generic algorithms (`sort`, `find`, etc)

STL

- C++ standard library provides a framework for dealing with data as sequences of elements, called STL (Standard Template Library).
- STL provides containers (`vector`, `list`, `map`, etc) and generic algorithms (`sort`, `find`, etc)
- Other standard library features, such as `ostream`, C-style string functions, are not part of STL.

Computation and Data

- There are two major aspects of computing:
  - the **computation**
  - the **data**

Computation and Data

- There are two major aspects of computing:
  - the **computation**:
    - if-statements
    - loops
    - functions
    - error-handling, etc.
  - the **data**:
    - `vectors`
    - `arrays`
    - `strings`
    - `files, etc`

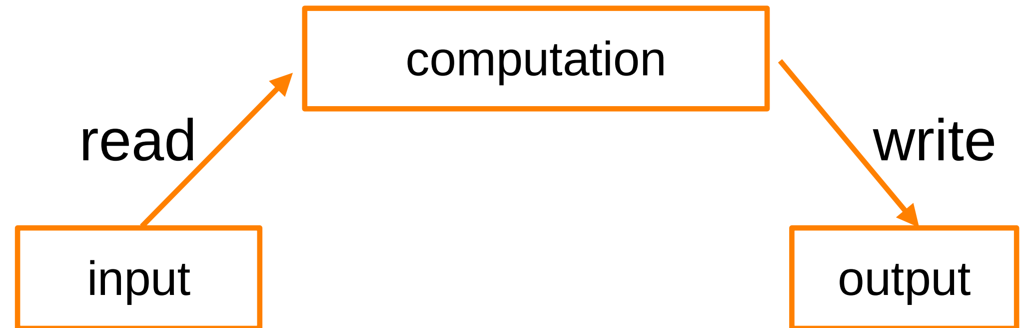
Computation and Data

- There are two major aspects of computing:
  - the **computation**:
    - if-statements
    - loops
    - functions
    - error-handling, etc.
  - the **data**:
    - `vectors`
    - `arrays`
    - `strings`
    - `files, etc`

Computation and Data

- There are two major aspects of computing:
  - the **computation**:
    - if-statements
    - loops
    - functions
    - Error-handling, etc.
  - the **data**:
    - `vectors`
    - `arrays`
    - `strings`
    - `files, etc`

- To do useful work, we need both.
  - a large amount of data is incomprehensible without analysis, visualizations, and searching for “the interesting bits”



Lots of data!

- We talk about lots of data!
  - dozens of Shapes
  - hundreds of temperature readings
  - thousands of log records
  - millions of points
  - billions of web pages, etc.
- We talk about processing containers of data, streams of data, etc.

Lots of data!

- We talk about lots of data!
  - dozens of Shapes
  - hundreds of temperature readings
  - thousands of log records
  - millions of points
  - billions of web pages, etc.
- We talk about processing containers of data, streams of data, etc.
- In particular, this is not a discussion of how best to choose a couple of values to represent a small object, such as a complex number, a temperature reading, or a circle.

Common tasks

To get an idea of what support we would like for writing our code, consider a more abstract view of what we do with data:

Common tasks

To get an idea of what support we would like for writing our code, consider a more abstract view of what we do with data:

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the Nth element)
  - By value (e.g., get the first element with the value "Chocolate")
  - By properties (e.g., get the first elements where “age<64”)
- Add data
- Remove data
- Sorting and searching
- Perform simple numeric operations (e.g., multiply all elements by 1.7)

Common tasks

We would like to do these things without getting sucked into a swamp of details about differences among containers, in ways of accessing elements, etc.

Looking back we can observe that we can (already) write programs that are similar independently of the data type used:

- Using an `int` isn't that different from using a `double`
- Using a `vector<int>` isn't that different from using a `vector<string>`

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a `vector` isn't all that different from finding a value in a `list` or an `array`
- Looking for a `string` ignoring case isn't all that different from looking at a `string` not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Ideals

We want to build on these observations to write a code that is

- easy to read
- easy to modify
- regular
- short
- fast

Ideals

To minimize our programming work we would like

- Uniform access to data
  - independently of how it is stored
  - independently of its type
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - retrieval of data
  - addition of data
  - deletion of data
- Standard versions of the most common algorithms
  - copy, find, search, sort, sum, …

Ideals

The STL library provides that, and more.

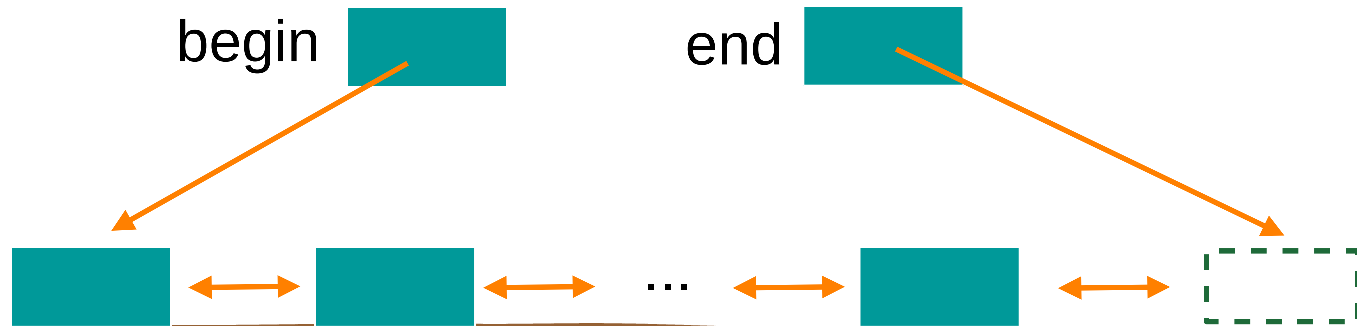
We will look at it not just a very useful set of facilities, but also as an example of a library designed for maximal flexibility and performance.

Sequences and Iterators

- The central concept of the STL is sequence.
  From the STL point of view, a collection of data is a sequence.

- A sequence has a beginning and an end, identified by a *pair of iterators*
  - *An iterator is an object that identifies an element of sequence*

- We can traverse a sequence from its beginning (*points to the first element – if any*) to its end (*points to the one-beyond-the-last element*), optionally reading or writing the value of each element.



Sequences and Iterators

- An iterator is a type that supports the “iterator operations”
  - ++ go to next element (++p)
  - * get value (*p)
  - == does this iterator point to the same element as that iterator? (p==q)
  - some iterators support more operations (e.g. --, +, and [ ])
- A pointer to an element of an array is an iterator
- However, many iterators are not just pointers (e.g., we can define an iterator that checks the range and throws an exception of we try to make it point outside its [begin:end) sequence or dereference end)

Basic model

- Iterators are used to connect our code (algorithms) to our data
- The writer of the code knows about the iterators (and not about the details of how the iterators actually get to the data),
- And the data provider supplies iterators rather than exposing details about how the data is stored to all users

Basic model

- Iterators are used to connect our code (algorithms) to our data
- The writer of the code knows about the iterators (and not about the details of how the iterators actually get to the data),
- And the data provider supplies iterators rather than exposing details about how the data is stored to all users

Algorithms

sort, find, search, copy, …

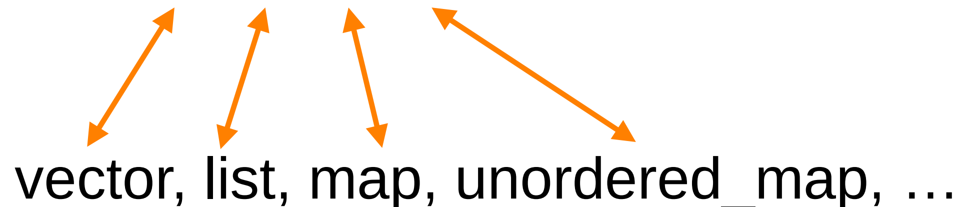
Containers

vector, list, map, unordered_map, …

Basic model

- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types
- **Algorithms**

sort, find, search, copy, …



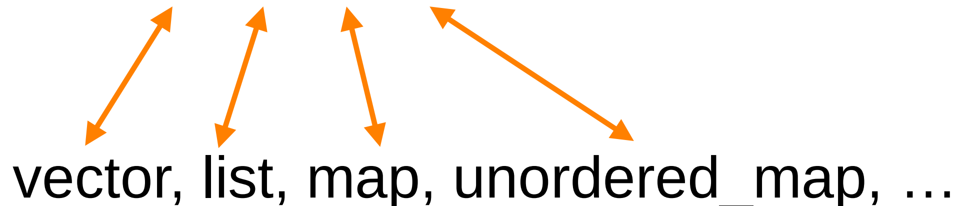
Containers

vector, list, map, unordered_map, …

Basic model

- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types
- **Algorithms**

  sort, find, search, copy, …



Containers

vector, list, map, unordered_map, …

To quote Alex Stepanov:
"The reason STL algorithms and containers work so well together is that they don't know anything about each other"

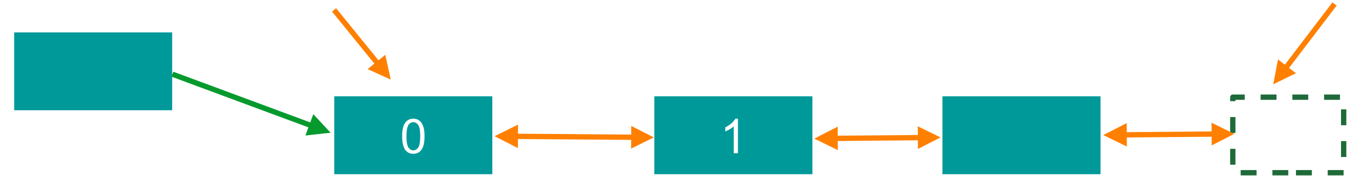
The STL

- The STL is an ISO (*International Organization for Standardization*) C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming

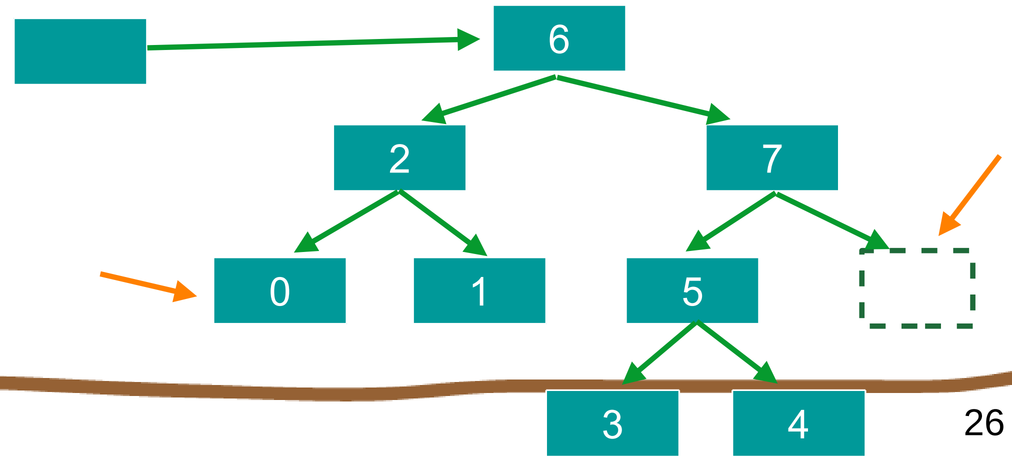
Containers
(hold sequences in different ways)

- vector

- list (doubly linked)



- set (a kind of a tree)



The simplest algorithm: find()

- Find the first element that equals a value

The simplest algorithm: find()

- Find the first element that equals a value

```
template<class It, class T>
It find(It first, It last, const T& val) {
   while (first != last && *first != val)
      ++first;
   return first;
}
```

The simplest algorithm: find()

- Find the first element that equals a value

```
template<class It, class T>
It find(It first, It last, const T& val) {
   while (first != last && *first != val)
      ++first;
   return first;
}

void f(vector<int>& v, int x) // find an int in a vector
{
   vector<int>::iterator p = find(v.begin(),v.end(),x);
   if (p != v.end()) { /* we found  x */ }
   // …
}
```

The simplest algorithm: find()

- Find the first element that equals a value

```
template<class It, class T>
It find(It first, It last, const T& val) {
   while (first != last && *first != val)
      ++first;
   return first;
}

void f(vector<int>& v, int x) // find an int in a vector
{
   vector<int>::iterator p = find(v.begin(),v.end(),x);
   if (p != v.end()) { /* we found  x */ }
   // …
}
```

We can ignore (“abstract away”) the differences between containers

find()
generic for both element type and container type

```
void f(vector<int>& v, int x)      // works for vector of ints
{   vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p! = v.end()) { /* we found  x */ }
    // …     }
```

find()

generic for both element type and container type

```
void f(vector<int>& v, int x)      // works for vector of ints
{   vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p! = v.end()) { /* we found  x */ }
    // …     }

void f(list<string>& v, string x)    // works for list of strings
{   list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p! = v.end()) { /* we found  x */ }
    // …     }
```

find()
generic for both element type and container type

```
void f(vector<int>& v, int x) // works for vector of ints
{  vector<int>::iterator p = find(v.begin(),v.end(),x);
   if (p! = v.end()) { /* we found  x */ }
   // …     }

void f(list<string>& v, string x)   // works for list of strings
{  list<string>::iterator p = find(v.begin(),v.end(),x);
   if (p! = v.end()) { /* we found  x */ }
   // …     }

void f(set<double>& v, double x) // works for set of doubles
{  set<double>::iterator p = find(v.begin(),v.end(),x);
   if (p! = v.end()) { /* we found  x */ }
   // …     }
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
  - ***not*** “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”

Simple algorithm: find_if()

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

Simple algorithm: find_if()

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

```
template<class It, class Pred>
It find_if(It first, It last, Pred pred) {
   while (first != last && !pred(*first))
      ++first;
   return first;
}
```

Simple algorithm: find_if()

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

```
template<class It, class Pred>
It find_if(It first, It last, Pred pred) {
   while (first != last && !pred(*first))
      ++first;
   return first;
}

void f(vector<int>& v) {
   vector<int>::iterator p = find_if(v.begin(),v.end(), Odd);
   if (p!=v.end()) { /* we found an odd number */ }
   // …    }
```

a predicate

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example:
  - A function

```
bool odd(int i) { return i%2; } // % is the remainder (modulo) operator
odd(7);                          // call odd: is 7 odd?
```

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example:
  - A function

```
bool odd(int i) { return i%2; } // % is the remainder (modulo) operator
odd(7);                         // call odd: is 7 odd?
```

  - A function object

```
struct Odd {
    bool operator()(int i) const { return i%2; }
};
Odd odd; // make an object odd of type Odd
odd(7);      // call odd: is 7 odd?
```

Predicates: more function objects

- Another example of function object , using state:

```
template<class T>
struct Less_than
{
   T val;   // value to compare with
   Less_than(const T& x) :val(x) { }  // constructor
   bool operator()(const T& x) const {
      return x < val;
   }
};
```

Predicates: more function objects

- Another example of function object , using state:

```
template<class T>
struct Less_than
{
    T val;   // value to compare with
    Less_than(const T& x) :val(x) { }  // constructor
    bool operator()(const T& x) const {
        return x < val;
    }
};

// find x < 43 in vector<int> :
p = find_if(v.begin(), v.end(), Less_than<int>(43));
```

Predicates: more function objects

- Another example of function object , using state:

```
template<class T>
struct Less_than
{
   T val;   // value to compare with
   Less_than(const T& x) :val(x) { }  // constructor
   bool operator()(const T& x) const {
      return x < val;
   }
};

// find x < "perfection" in list<string>:
q = find_if(ls.begin(), ls.end(), Less_than<string>("perfection"));
```

In-class work

- Let’s pause here, grab the file **functions.cpp** from our website, and do some in-class work that is at the end of the file

Function objects

- A very efficient technique
  - *Inlining* (when compiler tries to generate the code for the function at each point of call rather than using function-call instructions at run-time) very easy
    - and effective with current compilers
  - Faster than equivalent function
    - And sometimes you can’t write an equivalent function
- The main method of policy parameterization in the STL (*see next slide*)
- Key to emulating functional programming techniques in C++

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a “policy”.
  - Example: we need to parameterize sort by the comparison criteria

```
struct Record {
    string name;        // standard string for ease of use
    char addr[24];      // old C-style string to match database layout
    // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name()); // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr()); // sort by addr
```

Comparisons

```
// Different comparisons for Rec objects:

struct  Cmp_by_name {
   bool operator()(const Rec& a, const Rec& b) const
   { return a.name < b.name; }  // look at the name field of Rec
};

struct  Cmp_by_addr {
   bool operator()(const Rec& a, const Rec& b) const
   { return 0 < strncmp(a.addr, b.addr, 24); }   // correct?
};

// note how the comparison function objects are used to hide ugly
// and error-prone code
```

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
  - Example: we need to parameterize sort by the comparison criteria

```
vector<Record> vr;
// …
sort(vr.begin(), vr.end(),
    [] (const Rec& a, const Rec& b)
    { return a.name < b.name; } // sort by name
    );

sort(vr.begin(), vr.end(),
    [] (const Rec& a, const Rec& b)
    { return 0 < strncmp(a.addr, b.addr, 24); }  // sort by addr
    );
```

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
  - Example: we need to parameterize sort by the comparison criteria

```
vector<Record> vr;
// …
sort(vr.begin(), vr.end(),
    [] (const Rec& a, const Rec& b)
    { return a.name < b.name; }  // sort by name
    );

sort(vr.begin(), vr.end(),
    [] (const Rec& a, const Rec& b)
    { return 0 < strncmp(a.addr, b.addr, 24); }  // sort by addr
    );
```

called lambda-expressions

Policy parameterization

- Use a named object as argument
  - If you want to do something complicated
  - If you feel the need for a comment
  - If you want to do the same in several places
- Use a lambda expression as argument
  - If what you want is short and obvious
- Choose based on clarity of code
  - There are no performance differences between function objects and lambdas
  - Function objects (and lambdas) tend to be faster than function arguments

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook