COMP6771
Advanced C++ Programming

Week 2
Part 2: Arrays, STL Containers and Iterators

2016

www.cse.unsw.edu.au/~cs6771

Arrays

- In C, arrays are low-level constructs:

```
int a[10];   conceptually   int *b = a;
f(a);            ===>        f(b);
```

- C++
  - Use vector and list containers in STL whenever possible
  - Use C++11 arrays:

    ```
    std::array<int, 4> a = {0,1,2,3};
    std::array<int, 4> b = a;
    ```

  - An C++11 array know its own size and supports assignment, random access iterators, etc.

Caution

Both C and C++ style arrays need to know size at compile time.

Arrays

Arrays are data structures that store a collection of objects of the same type. C++ arrays are similar to C arrays.

An array:

- is an indexed data structure
- holds items of a single type
- has a fixed size
- must be sized at compile time
- may not hold reference types
- cannot be copied or assigned

NB

It is better to use a `vector` instead of an array, unless there is a good reason to use an array.

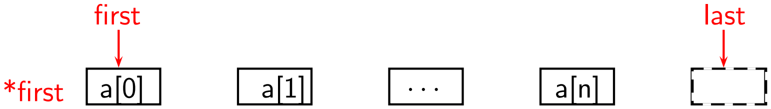
Iterating with Pointers

- Use array indices:

```
int a[10];
for (int i = 0; i < 10; ++i)
   do something on a[i]
```

- Use two pointers:

```
int *first = a;
int *last = a + 10;
for (; first != last; ++first)
   do something on *first
```



- last points to one past the last element
- int $*$ is an iterator for int[]!

Printing an 2D Array: C-Style

Pointer math!

```
#include <iostream>

int main() {
  int ia[3][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 10, 11, 12
  };

  for (int (*p)[4] = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
      std::cout << *q << ' ';
    std::cout << std::endl;
}
```

Printing an 2D Array: Type Aliases

Still pointer math!

```
#include <iostream>

using int_array = int [4]; // C++11
// or typedef int int_array[4];

int main() {
  int ia[3][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 10, 11, 12
  };

  for (int_array *p = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
      std::cout << *q << ' ';
    std::cout << std::endl;
}
```

Printing an 2D Array: `auto`

Cleaner but still some pointer math.

```
#include <iostream>

int main() {
  int ia[3][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 10, 11, 12
  };

  for (auto p = ia; p != ia + 3; ++p)
    for (auto q = *p; q != *p + 4; ++q)
      std::cout << *q << ' ';
    std::cout << std::endl;
}
```

Printing an 2D Array: Iterators

Easiest to read

```
#include <iostream>

int main() {
  int ia[3][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 10, 11, 12
  };

  for (auto p = std::begin(ia); p != std::end(ia); ++p)
    for (auto q = std::begin(*p); q != std::end(*p); ++q)
      std::cout << *q << ' ';
    std::cout << std::endl;
}
```

The Range-Based for in C++11: Use It!

- The syntax:

  ```
  for (declaration : expr)
    statement
  ```

  where `expr` is an object of a type representing a sequence.

- An example:

```
std::string s("Hello World!");
for (auto &c : s)
  c = std::toupper(c);
std::cout << s << std::endl; // HELLO WORLD!
```

Why Function Templates?

- As a strongly-typed language, C++ requires:

```
int min(int a, int b) { 1
  return a < b ? a : b;
}
double min(double a, double b) { // 2
  return a < b ? a : b;
}
... more for other types ...
```

- Call resolution due to function overloading:

```
min(1, 2); // 1
min(1.1, 2.2); // 2
...
```

What Are Function Templates?

- Definition:

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

- Uses:

```
min(1, 2)     // int min(int, int)
min(1.1, 2.2) // double min(double, double)
...
```

A function template is a prescription for the compiler to generate particular instances of a function varying by type

STL (Standard Template Library)

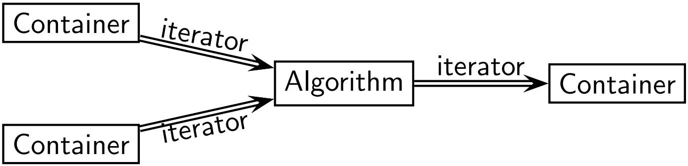
- Architecture and Design Philosophy
- Sequential Containers
- Associative Containers
- Iterators
- Generic Algorithms

From Week 5 onwards, we will learn how to

- write function templates, class templates and iterators, and
- deal with function objects and binders

Assignment 1 focuses on using streams and STL containers.

The STL Architecture



J Algorithms + K Containers = $J + K$ implementations
All containers/algorithms are templates and thus work for I types

- Algorithms manipulate data, but dont know about containers
- Containers store data, but dont know about algorithms
- Iterators are an abstraction of "pointer" or "index" - a means to access the appropriate element.
- Algorithms and containers interact through iterators
- Each container has its own iterator types

Example 1

- Read in an arbitrary number of integers ($n > 1$) from "numbers.txt" and display:
  - Minimum, maximum
  - Median
  - Average
  - Geometric mean $(y_1 * \cdots \times \ldots y_n)^{\frac{1}{n}}$
- How many lines would it take you?
- Arbitrary storage (for median), sorting, loops, ...

Example 1: a Solution based on STL

```
#include<math.h>
#include<iostream>
#include<fstream>
#include<algorithm>
#include<functional>
#include<numeric>
#include<vector>
#include<iterator>

int main() {
  std::vector<int> v;
  std::ifstream in("numbers.txt");
  std::copy(std::istream_iterator<int>(in),
            std::istream_iterator<int>(), std::back_inserter(v));

  std::sort(v.begin(), v.end());

  std::cout << "min/max: " << v.front() << " " << v.back() << std::endl;
  std::cout << "median: " << *(v.begin() + (v.size()/2)) << std::endl;
  std::cout << "average: " << accumulate(v.begin(), v.end(), 0.0) /
               v.size() << std::endl;

  std::cout << "geomean: " << std::pow(accumulate(v.begin(), v.end(),
               1.0, multiplies<double>()), 1.0/v.size()) << std::endl;
}
```

Example 2

Write a program that outputs the words and the number of times it occurs in a file (sorted by word)

```
#include <vector>
#include <map>
#include <ifstream>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<string> v;
  std::map<string, int> m;

  std::ifstream in("words.txt");
  std::copy(std::istream_iterator<string>(in),
     std::istream_iterator<string>(), std::back_inserter(v));

  for (auto vi = v.begin(); vi != v.end(); ++vi)
    ++m[*vi];

  for (auto mi = m.begin(); mi != m.end(); ++mi)
    std::cout << mi->first << ": " << mi->second << std::endl;
}
```

The Design Problem

Design a library operating on:

- I types: int, float, ...
- J containers: vector, list, map, ...
- K algorithms, search, find, sort, ...

- Naïve: $I \times J \times K$ implementations

- STL: $J + K$! Well, nearly so.

C++ STL

- The heart of the C++ standard library
- A generic (or reusable) library for managing collections of data with modern and efficient algorithms
  - Containers: vector, list, stack, ...
  - 100+ Algorithms, find, sort, copy, ...
  - Iterators are the glue between the two!
- An example of generic programming
- All components of the STL are templates
- For efficiency reasons, STL is not object-oriented:
  - Makes little use of inheritance, and
  - Makes no use of virtual functions

Nicolai M. Josuttis, "The C++ Standard Library: A Tutorial and Reference", Addison-Wesley, 2nd Edition, 2012. ISBN-10: 0-321-62321-5.

Example: Sorting and Printing Containers

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>

int main() {
  std::vector<int> v3, 2, 1;
  std::string s("string");

  std::sort(v.begin(), v.end());
  std::copy(v.begin(), v.end(),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;

  std::sort(s.begin(), s.end());
  std::copy(s.begin(), s.end(),
            std::ostream_iterator<char>(std::cout, " "));
  std::cout << std::endl;
}
```

Even `std::string` can be treated as a container.

Reasons to Use STL

- Code reuse, no need to re-invent the wheel.
- Efficiency; fast and use less resources. Modern C++ compilers are usually tuned to optimize for C++ standard library code.
- Accurate, less buggy.
- Terse, readable code; reduced control flow.
- Standardization, guaranteed availability.
- A role model of writing a library.
- Good knowledge of data structure and algorithms.

The STL Containers

Sequence Containers
- vector
- deque
- array C++11
- list
- forward_list C++11

Adaptors
- stack
- queue
- priority_queue

Associative Containers
- set
- map
- multiset
- multimap

Unordered Associative Containers
- unordered_set C++11
- unordered_map C++11
- unordered_multiset C++11
- unordered_multimap C++11

Container Abilities

Copyable/Movable/Assignable, ideally

```
std::vector<int> v1(5, 1); // 5 elements initialised with 1
std::vector<int> v2(v1);   // initialised: copied/moved from v1

v2 = v3;      // v2: (1) freed
              //     (2) assignment: copied/moved from v3
```

Will look at Copy Control in Week 3.

- Often rely on `operaor<` to be defined for the container elements
- Retraversing the same container yields the same order (provided you don't add and delete elements)
- Not safe in the sense that they check for every possible error. An operation will do the right things for you if you use it properly by meeting its requirements. Violating these requirements, such as using an invalid index, results in undefined behavior.

Container Operations (or Algorithms)

- Ideally, all operations should be supported by all containers
- In reality, not all operations are supported in all containers
  http://www.cplusplus.com/reference/stl/
- Common operations may have different time/space complexities – compare `vector` vs. `list`.
- To choose an appropriate container:
  - Which operations are required?
  - How often each operation will be performed (relatively)?

Container Efficiency: Guaranteed

Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

...

O(1)+: amortised constant time as the container may be resized

Writing Efficient Code in STL

- `size()` in `vector` is O(1)

```
vector<int> v(10, 1);
for (unsigned i = 0; i < v.size(); ++i)
  sum += v[i];
```

- `size()` is not provided in `forward_list`

```
forward_list<int> l(10, 1);
for (unsigned i = 0; i < l.size(); ++i) // error
  ...
```

The rationale behind can be found at:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2543.htm

Sequential Containers

- The programmer controls the order in which the elements are stored
- The order does not depend on the values of elements

Container	Access/Retrieval	Insert, Erase
`vector` (1D array)	O(1) random access	O(1) at back only O(N) at front, middle
`list` (doubly linked list)	O(1) at front/back only No random access (Would be O(N))	O(1) at any position
`forward_list` (singly linked list)	O(1) at front only No random access (Would be O(N))	O(1) at any position
`deque` (double-ended 1D array)	O(1) random access	O(1) at front/back only O(N) in middle

Fill the rest for `stack`, `queue` and `priority_queue`

Vector

- The most commonly used container is vector
  - Defined in the standard header <vector>
- Essentially, a vector is a dynamic array
  - Elements are sequential in memory
  - Fast random access to elements by index
  - Insertion and removal of elements at the end is fast
  - Insertion and removal elsewhere in the vector may be very slow

```
void read_and_print() {
  vector<int> numbers;
  int input;
  while (cin >> input)          // insert an element at
    numbers.push_back(input); // the end of the vector

  for (int n : numbers)
    cout << n << endl;
}
```

Vectors: Constructors

```
vector<int> v0 {1,2,3}; // a vector initialised as 1, 2 and 3
vector<int> v1;              // a vector of length 0
vector<float> v2(5, 1.0); // a vector of 5 elements,
                             // all initialised with 1.0
vector<double> v3(10);      // a vector of 10 elements,
                             // all initialised to the default 0.0
vector<double> v4(v2);      // a vector initialised as a
                             // a copy of v2
vector<string> words = { "Hello", "World" };
                             // a vector of two strings
int a = { 1, 2, 3 };
vector<int> v5(a, a + 3); // a vector of 3 elements
                             // initialised from a
vector<int> v6(v2.begin(), v2.end());
                             // a vector initialised with any
                             // two iterators defining a range

vector<noDefaultCtor> v7(10); // error
```

A Variety of Vectors

```
vector<int*> v1;


vector<vector<int>> v2;


vector<map<string, list<int>>> v3;


vector<list<string>*> v4;
```

Accessing Vectors

Vectors have random access like arrays:

- [] can be used to access elements without any run-time check of the index (result is undefined if the index is invalid!)
- The member function at() provides a safer (but slower) version of [], which throws an exception if the index is illegal

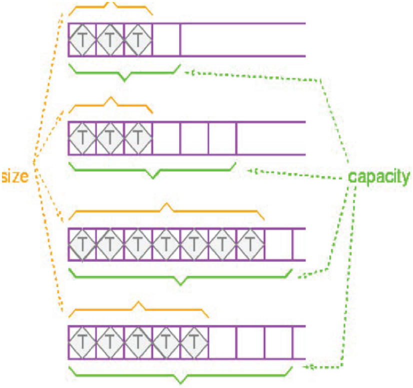
```
vector<string> words = { "Hello", "World" };

for (unsigned i = 0; i < words.size(), ++i)
   cout << words[i] << endl;

try {
   cout << words.at(2) << endl;
} catch(const std::out_of_range &e) {
   cout << e.what() << endl;
}
```

Size and Capacity

- A vector can be presized, supplying the size at construction, and you can ask a vector how many elements it has with size(). This is the logical number of elements.
- There is a also a notion of capacity() – the number of elements the vector can hold before reallocating.




```
vector<int> v;
cout << v.size( << " " << v.capacity() << endl; // 0 0
v[0] = 1; // runtime error: segmentation fault
v.push_back(1);
cout << v.size( << " " << v.capacity() << endl; // 1 1
```

Using assign

```
vector<int> v1(10);
vector<float> v2 = v1; // error: type mismatch
                       // requirements:
                       // (1) v1 and v2 have the same type
                       // (2) their elements have the same type

v2.assign(v1.begin(), v1.end());
                       // ok
                       // but the elements in v1 must be
                       // type-assignable to v2

list<int> v3(20, 0);
v2.assign(v3.begin(), v3.end());  // ok

list<string> v4(20, "Hi");
v2.assign(v4.begin(), v4.end());  // error
```

Insertion and Removal

- Elements can be efficiently inserted and removed from the end of the vector

```
vector<string> words = { "Hello", "World" };

words.push_back("C++"); // { "Hello", "World", "C++" }

words.pop_back(); // { "Hello", "World" }
```

- Elements can also be inserted and erased from any place in the vector, though inefficiently

```
vector<string> words = { "Hello", "World" };

vector<string>::iterator i = words.begin();
words.insert(++i, "Lovely");
             //  { "Hello", "Lovely", "World" };

words.erase(words.begin());

             //  { "Lovely", "World" };
```

Other Sequential Containers

- std::list is implemented as a doubly-linked list:
  - No methods for random (index-based) access
  - There is support for fast merging and splicing of lists

```
list<string> words = { "Hello", "World" };
list<string> words2 = { "Lovely" };

words.splice(++words.begin(), words2);
          // words: { "Hello", "Lovely", "World" };
          // words2: empty
```

- std::forward_list is a singly-linked list with pointers from each node only to the next one
  - Takes less memory
  - But has a slimmer interface, e.g., no -- for iterators

```
list<string> words = { "Hello", "World" };
list<string> words2 = { "Lovely" };

words.splice_after(++words.begin(), words);
          // The results are the same as above

```

Other Sequential Containers

- std::deque is like a vector, but allows fast insertion and removal from its beginning as well
  - Using push_front() and pop_front()

```
deque<string> words = { "Hello", "World" };

words.pop_front();       // {  "World" };
words.push_front("Hi"); // { "Hi", "World" };
```

- std::array is a constant-size array introduced in C++11:
  - Size must be known at compile time
  - Basically wraps a C-style array inside

```
array<string, 2> words = { "Hello", "World" };

words.at(0) = "Hi";        // { "Hi", "World" };
cout << words.at(3) << endl; // throws an out-of-range error
```

Adaptors: `stack`, `queue` and `priority_queue`

- STL uses adaptors to create new data structures from existing containers, using composition and delegation:
  - defines `stack`, `queue` and `priority_queue` as adaptors built from a basic sequence type
  - but with the API that we really want (e.g., `push()` and `pop()`)
- Programmers are encouraged to create their own adaptor classes based on STL containers.

Associative Containers

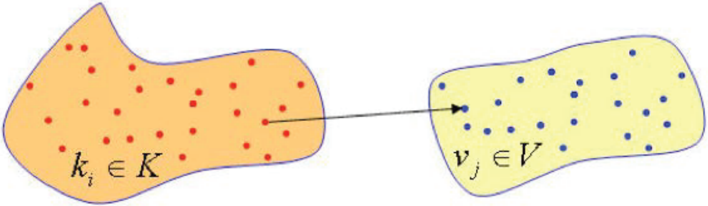
- A value type is accessed through a second data type, the key
- The elements are either unordered or ordered based on their key values rather than the order of insertion
- By default, operator< is used to compare the keys

Set

- For a set, the value type and key are the same

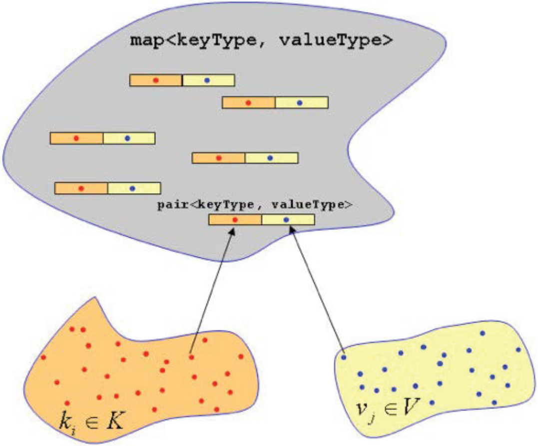
```
set<int> intSet;
s.push_front(1);
s.push_front(2);
s.push_front(1); // error: can't have duplicates

set<Sales_data> s;
s.push_front(Sales_data("123", 2, 33.0));
// error: operator< is not found in Sales_data
```



map

- A collection of pairs



An Example on `map`

```
#include <iostream>
#include <string>
#include <map>

int main() {
  std::map<std::string, double> m;
  std::pair<std::string, double> p1("bat", 14.75);
  std::pair<std::string, double> p2("cat", 10.157);
  std::pair<std::string, double> p3("dog", 43.5);
  m.insert(p1);
  m.insert(p2);
  m.insert(p3);

  for(auto mit = m.begin(); mit!=m.end(); ++mit)
    std::cout << mit->first << ' ' << mit->second << std::endl;
}
```

Querying `map` for Elements

- Intuitive method (lookup via indexing)

```
std::map<std::string, int> words;
if (words["bach"] ==  0)
   // bach not present
```

but the key has been inserted into the map if already present!

- Alternatively, can use map's `find()` operation to return an iterator pointing the queried key/value pair.

```
std::map<std::string, int>::iterator it;
it = words.find("bach");
if (it ==  words.end())
   // bach not present
```

Other Associative Containers

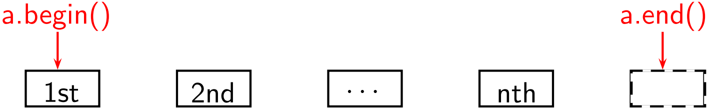
- The STL supplies multiset and multimap which allow storing the same key multiple times
- In addition, C++11 adds unordered_set, unordered_map, unordered_multiset and unordered_multimap, implemented using hash-tables instead of binary trees

Iterators

- Iterators that are classes (or types) support an abstract model of data as a sequence of objects
- An iterator is an abstract notion of pointers
- Glue between containers and generic algorithms:
  - The designer of algorithms do not have to be concerned with details about various data structures
  - The designer of containers do not have to provide extensive access operations

An Iterator for a Container

- a is a container with all its *n* objects ordered



- a.begin(): a "pointer" to the first element
- a.end(): a "pointer" to one past the last element
- if p "points" to the k-th element, then
  - *p is the object pointed to
  - ++p "points" to $(k+1)$-st element
- The loop:

```
for (first = a.begin(); first != a.end(); ++first)
   do something on *first
```

Terminology

- Iterators represent an abstract notion of pointers
- Every iterator declared is non-const, so that it can be used to iterate through the elements in a container
- A const iterator for a container $\equiv$ an iterator that "points' to a const element in the container
  $\Longrightarrow$ cannot use the iterator to modify the container elements
- A non-const iterator for a container $\equiv$ an iterator that "points' to a non-const element in the container
  $\Longrightarrow$ can use the iterator to modify the container elements

Recall const and non-const references!

Iterators for non-const Containers

```
vector<int> x(10, 1); // similarly for list<int> x(10, 1);
```

- Pre-C++11: asks for a non-const iterator explicitly

```
for(std::vector<int>::iterator first = x.begin();
    first != x.end(); ++first) {
  cout << *first << endl;
  *first = 10; // ok
  }
// can also ask for a const iterator
```

- C++11:

```
// asks for a non-const iterator to be inferred
for(auto first = x.begin(); first != x.end(); ++first) {
  cout << *first << endl;
  *first = 10; // ok
  }
// asks for a const iterator to be inferred
for(auto first = x.cbegin();
first != x.cend(); ++first) {
  cout << *first << endl;
  *first = 10; // error
  }
```

Iterators for const Containers

```
const vector<int> x(10, 1); // const list<int> x(10, 1);
```

- Pre-C++11: asks for a const iterator explicitly

```
for(std::vector<int>::const_iterator first = x.begin();
    first != x.end(); ++first) {
  cout << *first << endl;
  }
```

- C++11:

```
// a const iterator is automatically inferred
for(auto first = x.begin(); first != x.end(); ++first) {
  cout << *first << endl;
  }
```

- Cannot modify any container element:

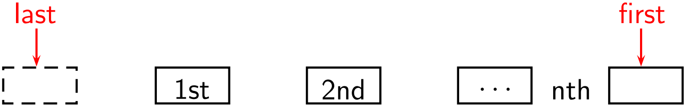
```
*first = 10; // error
```

Cannot obtain a non-const iterator from a const container

Reverse Iterators for a Container

```
std::vector<int> x(10, 1);
// non-const iterator
for(auto first = x.rbegin(); first != x.rend(); ++first) {
    cout << *first++ << endl;
  }

// const iterator
for(auto first = x.crbegin(); first != x.crend(); ++first) {
    cout << *first++ << endl;
  }
```



- first: a "pointer" to the last element
- last: a "pointer" to one past "one pass the first element"
- if p "points" to the k-th element, then
    - *p is the object pointed to
    - ++p "points" to $(k-1)$-st element

Five Categories of Iterators

Operation	ITERATORS				
	OUTPUT	INPUT	FORWARD	BI-DIR	RANDOM
Read		`=*p`	`=*p`	`=*p`	`=*p`
Access		`->`	`->`	`->`	`-> []`
Write	`*p=`		`*p=`	`*p=`	`*p=`
Iteration	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Compare		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

Different algorithms require different kinds of iterators for their operations:

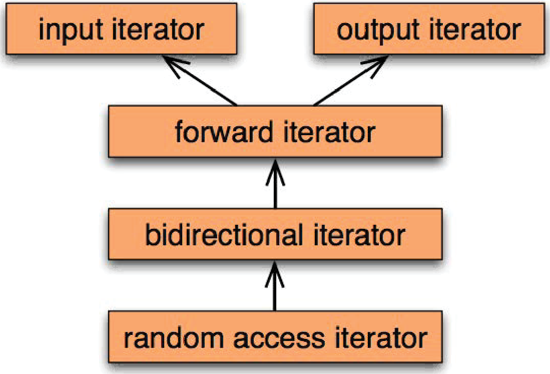
- input: `find(), equal(), ...`
- output: `copy()`
- forward: `replace(), ...`
- bi-directional: `next_permutation(), reverse(), ...`
- random: `sort, binary_search(), nth_element(), ...`

The Iterator Categories for STL Containers

CONTAINER	ITERATOR CATEGORY
vector	random access
deque	random access
list	bi-directional
forward_list	forward
stack	no
queue	no
priority_queue	no
map/multimap/set/multiset (ordered & unordered)	bi-directional

In Assignment 3, you will implement iterators for class templates.

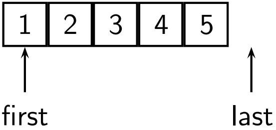
Class Hierarchy



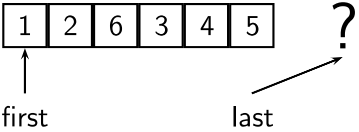
- Iterators are increasingly more powerful downwards
- The "$\rightarrow$: subtyping relation
- E.g., an algorithm that works for an input iterator should work nicely with a bi-directional iterator

Iterator Invalidation

```
vector<int> v{1, 2, 3, 4, 5};
auto first = v.begin();
auto last = v.end();
```

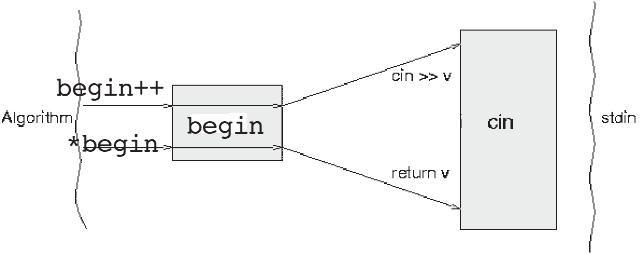



```
v.insert(find(v.begin(), v.end(), 3), 6);
```



- first is valid (unless reallocation happens) but last is not
- Ok for list or forward_list unless either pointer pointed to $\boxed{3}$
- Read the spec of each algorithm for detail!

istream_iterator

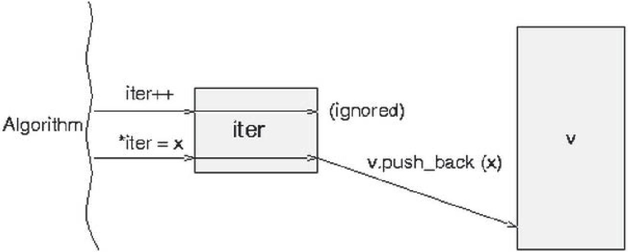



```
std::ifstream in(‘‘data.in’’);

std::istream_iterator<int> begin(in);
std::istream_iterator<int> end;
std::cout << *begin++ << std::endl; // read the first int
++begin;                            // skip the 2nd int
std::cout << *begin++ << std::endl; // read the third int

while (begin != end) {
  std::cout << *begin++ << std::endl; // read and print the rest
}
```

back_insert_iterator



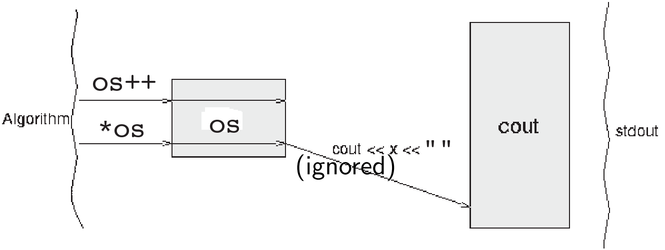

```
std::ifstream in(``data.in'');

std::istream_iterator<int> begin(in);
std::istream_iterator<int> end;

std::vector<int> v;
std::back_insert_iterator<std::vector<int>> iter(v);

while (begin != end)
  *iter++ = *begin++;
```

ostream_iterator




```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
 std::vector<int> v{1, 2, 3, 4, 5};
 std::ostream_iterator<int> os(std::cout, " ");
 for (const auto &i : v)
    *os = i; // or *os++ = i;
}
```

Algorithms

- 100+ generic algorithms, which operate on some expected iterator categories
  http://www.cplusplus.com/reference/algorithm/
- Defined in header <algorithm>
- We have already seen std:sort
- std::copy

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
    OutputIterator target) {
  while (first != last)
    *(target++) = *(first++);

  return result;
}
```

Container-Specific Operations Preferred

- Prefer container-specific over generic operations in `algorithm`
- List-specific operations can swap its elements by changing the links instead of swapping the values of those elements we swap links In linked lists,

	These operations return `void`.
`lst.merge(lst2)` `lst.merge(lst2, comp)`	Merges elements from `lst2` onto `lst`. Both `lst` and `lst2` must be sorted. Elements are removed from `lst2`. After the merge, `lst2` is empty. The first version uses the < operator; the second version uses the given comparison operation.
`lst.remove(val)` `lst.remove_if(pred)`	Calls `erase` to remove each element that is `==` to the given value or for which the given unary predicate succeeds.
`lst.reverse()`	Reverses the order of the elements in `lst`.
`lst.sort()` `lst.sort(comp)`	Sorts the elements of `lst` using < or the given comparison operation.
`lst.unique()` `lst.unique(pred)`	Calls `erase` to remove consecutive copies of the same value. The first version uses `==`; the second uses the given binary predicate.

Criticisms

- Cryptic error messages
- Careless use of STL templates can lead to code bloat
- STL containers are not intended to be used as base classes as their destructors are deliberately non-virtual
- Using invalid iterators is a common source of errors
- ...

Readings

- Chapters 9 – 11
  - But ignore the sections on binders and function objects, which will be covered later
- C++ Reference
  http://www.cplusplus.com/reference/stl/
  http://www.cplusplus.com/reference/algorithm
- SGI Standard Template Library Programmer's Guide
  http://www.sgi.com/tech/stl/

Next Lecture: Classes