The Standard C++ Library

Version 1: Dr. Ofir Pele
Version 2: Dr. Erel Segal-Halevi

Main Ideas

- **Purpose**
- **Flexibility**
- **Efficiency**
- **Simple & Uniform Interface**

Q: What types can we put in a template param?

A: Type which models the requested concept**.**

Concepts - Example

- Consider:

```
template<typename T> const T& min(const T& x,
                      const T& y)
{
   return x < y? x : y;
}
```

- The user must provide a type that has a less-than operator (<) to compare two values of type T, the result of which must be convertible to a Boolean value

Concepts - Example

- Consider:

```
template<typename T> const T& min(const T& x,
                      const T& y)
{
   return x < y? x : y;
}
```

- T
t
t
t

The problem:
cryptic error messages
from the implementation of the function
instead of a clear error message

ss-

ible

Concepts – What we would like it to be:

- **Not C++ syntax**:

```
template<LessThanComparable T> const T& min(const T& x,
                                            const T& y)
{
   return x < y? x : y;
}
```

- The user must provide a type that has a less-than operator (<) to compare two values of type T, the result of which must be convertible to a Boolean value

Concepts

- Concepts are not a yet part of the C++ language,

- Currently there is no (standard) way to declare a concept in a program, or to declare that a particular type is a model of a concept

- “were not ready” for C++11,C++14,C++17

- C++-20 ?

Concepts - advanced

- gcc has a new mechanism for concept checking.

To activate it define: **_GLIBCXX_CONCEPT_CHECKS**

Warning: not clear if it helps or not.

- Boost library has a Concept Check Library:

http://www.boost.org/libs/concept_check/concept_check.htm

```
template<typename T> const T& min(const T& x,
                  const T& y)
{
   BOOST_CONCEPT_ASSERT((LessThanComparable <T>));
   return x < y? x : y;
}
```

Concepts

- A concept is a list of requirements on a type.
- STL defines a hierarchy of concepts for containers, iterators, and element types.
- Concepts for element types include:

Equality Comparable -
types with operator== ,...

LessThan Comparable -
types with operator< ,...

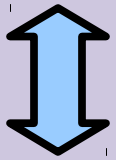
Assignable -
types with operator=
and copy Ctor

Concepts

- Cleanly separate interface from implementation.
- Primitives can also conform to a concept.

Concepts refinement

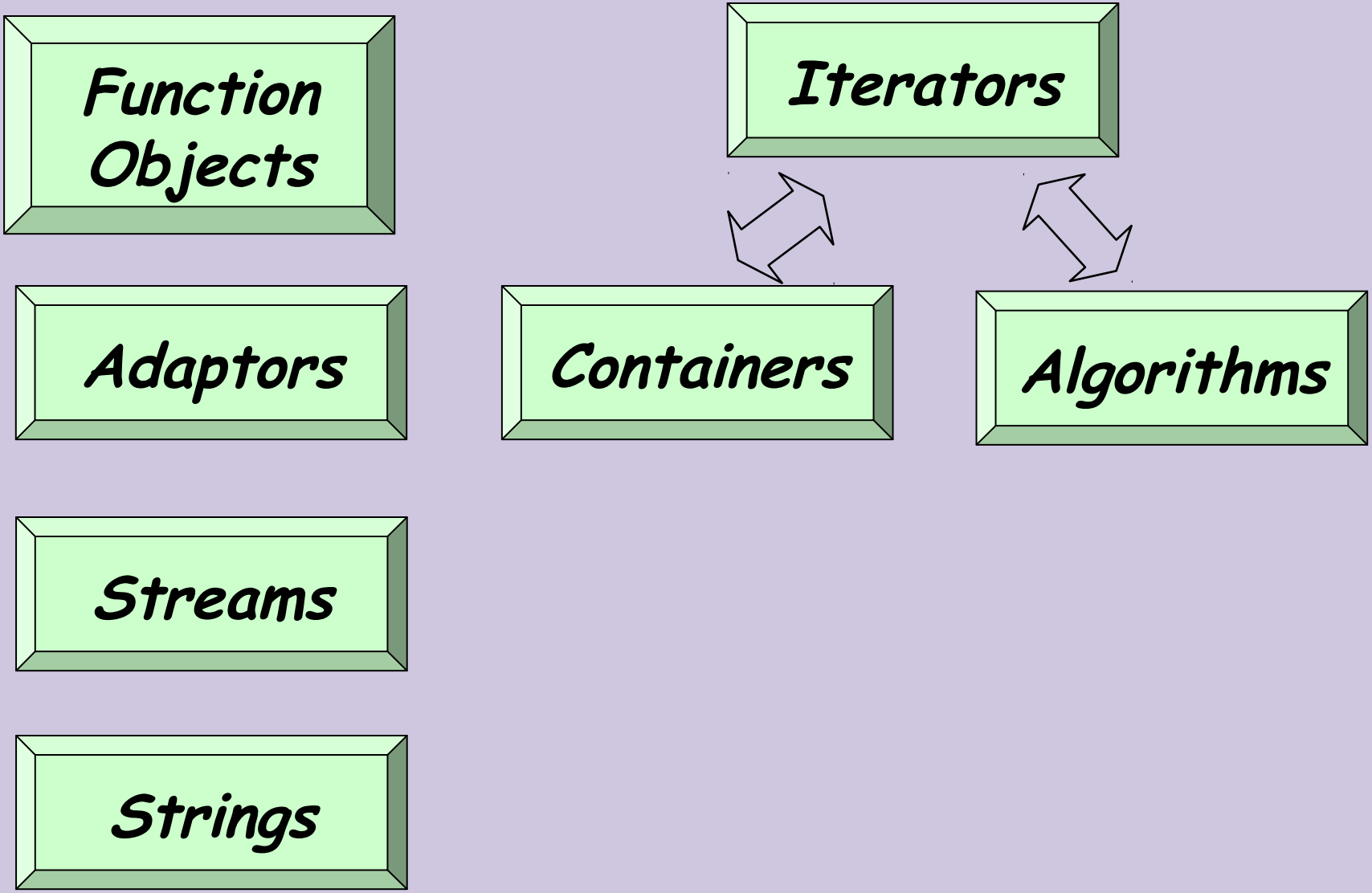
Concept B **is a refinement of concept** A



Concept B **imposes some additional requirements** on A

- Similar to inheritance.

Main Components



Containers

- Holds **copies** of elements.
- **Assumes** elements have:
  Copy Ctor & operator =
- The standard defines the **interface**.
- Two main classes
  - **Sequential containers**:
    list, vector,....
  - **Associative containers:**
    map, set ...

Assignable -
types with operator=
and copy Ctor

Containers documentation

see
http://www.cplusplus.com/reference/stl/

Sequential Containers

- Maintain a linear sequence of objects

Sequential Containers

list - a linked list of objects

- Efficient insertion/deletion in front/end/middle

vector - an extendable sequence of objects

- Efficient insertion/deletion at end
- Random access

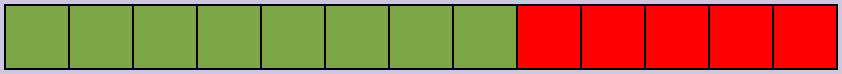
deque – double-ended queue

- Efficient insertion/deletion at front/end
- Random access

forward_list

array

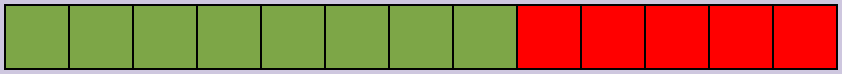
vector<T>



- Contiguous array of elements of type T
- Random access
- Can grow on as needed basis

```
std::vector<int> v(2);
v[0]= 45;
v[1]= 32;
v.emplace_back(60); //C++11
```

vector<T>



- Contiguous array of elements of type T
- Random access
- Can grow on as needed basis

```
std::vector<int> v(2);
v[0]= 45;
v[1]= 32;
v.push_back(60);// old style,
  // we will talk about
// the difference
```

emplace_back / push_back Average Time Complexity

If we inserted **n** elements we paid:

1+**2**+1+**4**+1+1+1+**8**+…+**n** =

O(n) + **1+2+4+…+n**

1+2+4+…+n = ?

emplace_back / push_back Average Time Complexity

$$1+2+4+\ldots n = a$$

emplace_back / push_back Average Time Complexity

$$1+2+4+\ldots n = a$$

$$1+2+4+\ldots n+2n = a+2n$$

emplace_back / push_back Average Time Complexity

$$1+2+4+\ldots n = a$$

$$1+2+4+\ldots n+2n = a+2n$$

$$1+2(1+2+\ldots+ n)=a+2n$$

emplace_back / push_back Average Time Complexity

$$1+2+4+\ldots n = a$$

$$1+2+4+\ldots n+2n = a+2n$$

$$1+2(1+2+\ldots+ n)=a+2n$$

$$1+2(a)=a+2n$$

emplace_back / push_back Average Time Complexity

$$1+2+4+\ldots n = a$$

$$1+2+4+\ldots n+2n = a+2n$$

$$1+2(1+2+\ldots+ n)=a+2n$$

$$1+2(a)=a+2n$$

$$a=2n-1$$

emplace_back / push_back Average Time Complexity

If we inserted **n** elements we paid:

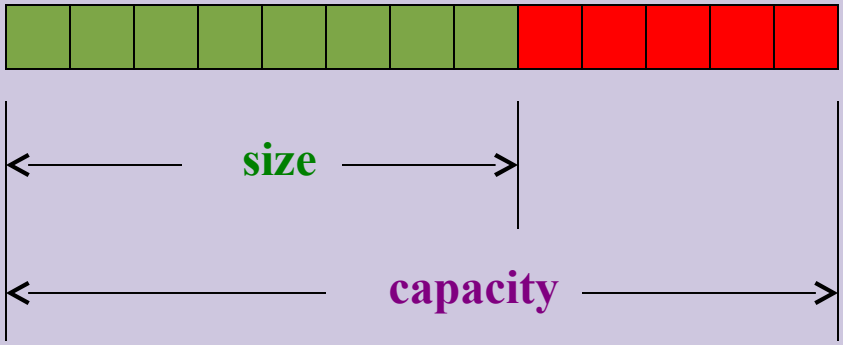
1+**2**+1+**4**+1+1+1+**8**+…+**n** =

O(n) + **1**+**2**+**4**+…+**n** =

O(n)

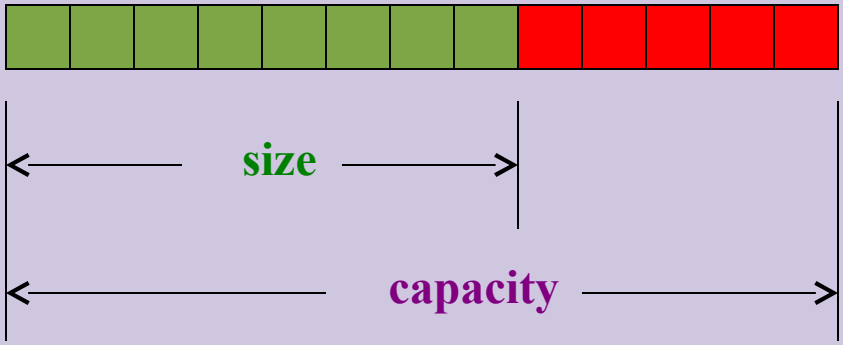
On average an each insertion cost O(1)

size and capacity



- The first “size” elements are constructed (initialized)
- The last “capacity - size” elements are uninitialized

size and capacity



- size_type **size**() const
- size_type **capacity**() const

C++ vs. Java

- Look at cplusplus documentation of vector.
- Look at Java documentation of Vector.
- Differences:
  - Simple class vs. interface and vtable.
  - Simple elements vs. class element.
  - Two accessors (with and without range check) vs. a single accessor

Creating vectors

- Empty vector:

```
std::vector<int> vec;
```

- vector with 10 ints each one of value int()==0:

```
std::vector<int> vec(10);
std::vector<int> vec(10,0); // better
```

- vector with 10 ints with the value of 42:

```
std::vector<int> vec(10, 42);
```

Creating vectors

- Empty vector:

```
std::vector<int> vec;
```

- vector with 10 ints each one of value **int()==0**:

```
std::vector<int>    vec(10);
std::vector<int>    vec(10,0); // better
```

Notice: int() is NOT a default constructor of int. Uninitialized ints (int k;) contain junk.

Creating vectors

- Empty vector:

```
std::vector<Fred> vec;
```

- vector with 10 default constructed Fred objects. i.e: 10 Fred() objects:

```
std::vector<Fred> vec(10);
```

- vector with 10 non-default constructed Fred objects:

```
std::vector<Fred> vec(10, Fred(5,7));
```

Creating vectors: C++11

- vector with different ints inside it:

```
std::vector<int> vec{1, 5, 10, -2, 0, 3};
```

- vector with different Fred objects:

```
std::vector<Fred> vec{Fred(5,7), Fred(2,1) }
```

Or

```
std::vector<Fred> vec{      {5,7},      {2,1} }
```

Associated types in vector

vector<typename T>::

- value_type - The type of object, T, stored
- reference - Reference to T
- const_reference - const Reference to T
- iterator - Iterator used to iterate through a vector *(how would you write it?)*
- …
-

[break in first week]

Time complexity

- Random access to elements.
- Amortized constant time insertion and removal of elements at the end.
- Linear time insertion and removal of elements at the beginning or in the middle.
- vector is the simplest of the STL container classes, and in many cases the most efficient.

Adding elements

- Inserts a new element at the end:

  void push_back(const T&)

  - **a.push_back(t)**
    - equivalent to:
  - **a.insert(a.end(), t)**

  } amortized constant time

- **insert** is linear time in the beginning or in the middle.

Adding elements-C++11

- Construct and insert a new element at the end:

  template<typename… Args>

  void emplace_back(Args&&… args)

  - **a.emplace_back(t)**
  - equivalent to:
  - **a.emplace(a.end(), t)**

  } amortized constant time

- **emplace** is linear time in the beginning or in the middle.

Accessing elements

Without boundary checking:

- **reference operator[](size_type n)**
- **const_reference operator[](size_type n) const**

With boundary checking:

- **reference at(size_type n)**
- **const_reference at(size_type n) const**

What about checking boundaries only in DEBUG mode? - Linux

- g++ has a standard library in DEBUG mode, to activate it define _GLIBCXX_DEBUG

  (g++ -D_GLIBCXX_DEBUG …)

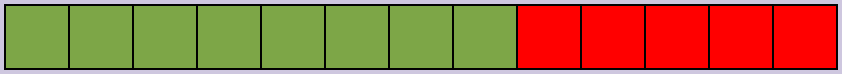
- stlport is an implementation of the standard library which includes DEBUG mode (havn't checked it myself yet):

http://www.stlport.org/

What about checking boundaries only in DEBUG mode? - MS

- In MSVS 2012 Debug build is automatically safe and Release build mode is not
- Other versions also have these kind of “Safe STL” modes but might require defining some flags to turn off or on.

vector<T>

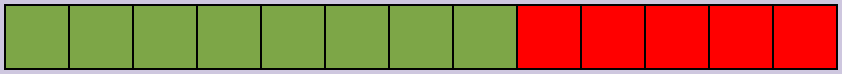


- **Contiguous** array of elements of type T
- We can get the underlining T* pointer:
  - if size()>0

  T* p= &(v[0])
  - c++11:

  T* p= v.data()

vector<T>



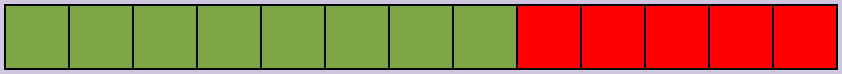
- **Contiguous** array of elements of type T
- We can get the underlining T* pointer:
  - if size()>0

    T* p= &(v[0])
  - c++11:

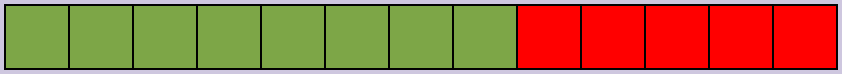
    T* p= v.data()

Useful for interfacing with C
or wrappers that work with C
like Matlab's mex

vector<T> v

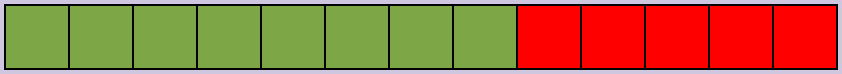


vector<T> v

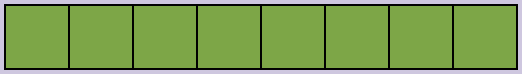


v.shrink_to_fit() // c++11

vector<T> v



v.shrink_to_fit() // c++11



or

Associative Containers

- Supports efficient retrieval of elements (values) based on keys.

- (Typical) Implementation:

red-black binary trees

hash-table (added in c++11)

Sorted Associative Containers

set

- A set of unique keys ordered by <

map

- Associate a value to key (associative array)
- Unique value of each key, ordered by <

multiset, multimap

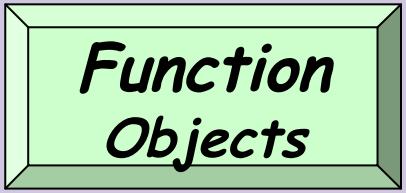
- Same, but allow multiple values

unordered_set, unordered_map

- Same, but without order (faster).

Sorted Associative Containers & Order

- Sorted Associative containers use operator< as default order
- We can control order by using our own comparison function
- To understand this issue, we need to use **function object**



Function Objects

Anything that can be called as if a function.

For example:

- Pointer to function
- A class that implements `operator()`
- Lambda expressions (c++11)

Example (folder 2)

```
class c_str_less {
public:
   bool operator()(const char* s1,
                   const char* s2) {
     return (strcmp(s1,s2) < 0);
   }
};

c_str_less cmp;  // declare an object
if(cmp("aa","ab"))
    …
if( c_str_less()("a","b") )
```

Creates temporary objects, and then call operator()

Template comparator example

```
template<typename T>
class less {
public:
  bool operator()(const T& lhs, const T& rhs)
  { return lhs < rhs; }
};

  less<int> cmp;  // declare an object
  if( cmp(1,2) )
     …
  if( less<int>()(1,2) )
     …
```

Creates temporary objects, and then call operator()

Using Comparators

```
// ascending order
// uses operator < for comparison
set<int>  s1;
set<int,less<int>> s1; // same

// descending order
// uses operator > for comparison
set<int,greater<int>> s2;
```

Using Comparators

```
set<int,MyComp> s3;
```

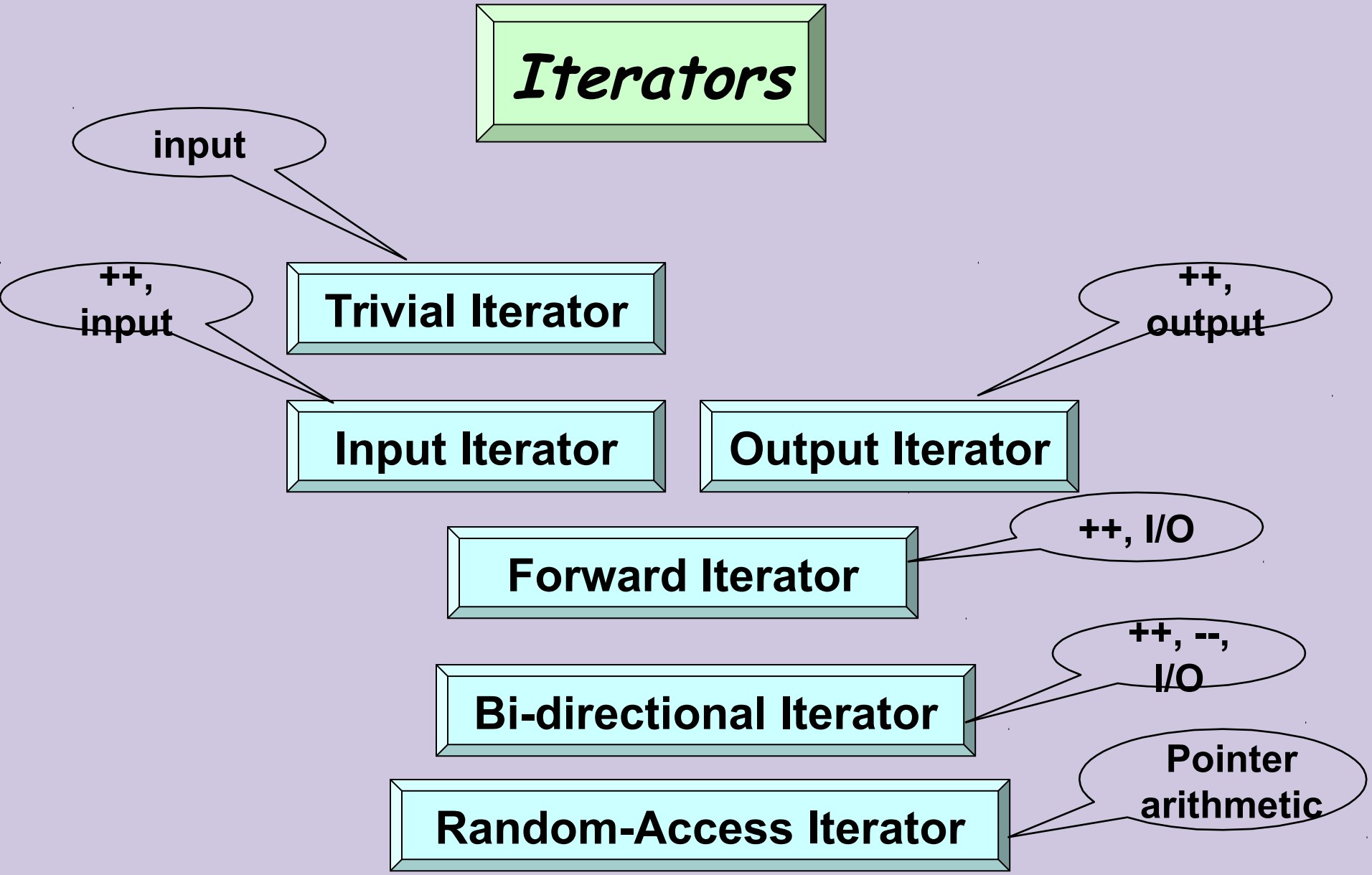
Creates a default constructed MyComp object.

```
MyComp cmp(42);
set<int,MyComp> s4(cmp);
```

Use given MyComp object.

Why should we use classes as function objects?

- So we get the “power” of classes.
- Examples:
  - Inheritance.
  - To parameterize our functions in run time or in compile time.
  - To accumulate information.


Iterators
input
Trivial Iterator
++, input
++, output
Input Iterator
Output Iterator
++, I/O
Forward Iterator
++, --, I/O
Bi-directional Iterator
Pointer arithmetic
Random-Access Iterator

Iterator Types

	Output	Input	Forward	Bi-directional	Random
Read		`x = *i`	`x = *i`	`x = *i`	`x = *i`
Write	`*i = x`		`*i = x`	`*i = x`	`*i = x`
Iteration	`++`	`++`	`++`	`++, --`	`++, --, +, -, +=, -=`
Comparison		`==, !=`	`==, !=`	`==, !=`	`==, !=, <, >, <=, >=`

- Output: write only and can write only once
- Input: read many times each item
- Forward supports both read and write
- Bi-directional support also decrement
- Random supports random access

(just like C pointer)

Iterators & Containers

Input/output/forward iterators:

- iostreams (folder 3)

Bidirectional iterators:

- list, map, set

Random access iterators:

- vector

Iterators & Containers

```
class NameOfContainer {
   …
   typedef … iterator; // iterator type
   iterator begin();      // first element
   iterator end();        // element after last
```

```
NameOfContainer<...> c

…

NameOfContainer<...>::iterator it;

for( it= c.begin(); it!=c.end(); ++it)

      // do something that changes *it
```

Iterators & Containers: c++11

```
class NameOfContainer {
   …
   typedef … iterator; // iterator type
   iterator begin();      // first element
   iterator end();        // element after last
```

```
NameOfContainer<...> c

…


for(auto it= c.begin(); it!=c.end(); ++it)
      // do something that changes *it
```

Iterators & Containers: c++11

```
class NameOfContainer {
   …
   typedef … iterator; // iterator type
   iterator begin();      // first element
   iterator end();        // element after last
```

```
NameOfContainer<...> c

…


for(auto& val : c)
      // do something that changes val
```

const_iterators & Containers

```
class NameOfContainer {
   …
typedef … const_iterator; // iterator type
const_iterator begin() const;      // first element
const_iterator end() const;        // element after last
```

```
NameOfContainer<...> c
…
NameOfContainer<...>::const_iterator it;
for( it= c.begin(); it!=c.end(); ++it)
      // do something that does not change *it
```

const_iterators & Containers: c++11

```
class NameOfContainer {
   …
typedef … const_iterator; // iterator type
const_iterator cbegin() const;      // first element
const_iterator cend() const;        // element after last
```

```
NameOfContainer<...> c

…


for(auto it= c.cbegin(); it!=c.cend(); ++it)
      // do something that does not change *it
```

const_iterators & Containers: c++11

```
class NameOfContainer {
   …
typedef … const_iterator; // iterator type
const_iterator cbegin() const;      // first element
const_iterator cend() const;        // element after last
```

```
NameOfContainer<...> c

…


for(const auto& val : c)
      // do something that does not change val
```

const_iterators & Containers

```
...
const_iterator cbegin() const;
const_iterator cend() const;
const_iterator begin() const;
const_iterator end() const;
```

```
...

iterator begin();
iterator end();
```

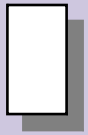
Note that the begin() and end() methods that return regular iterator are not **const** methods. i.e: if we get a container by const (const ref, ...) we can't use these methods. We have to use the methods that return **const_iterator**

[end of first week]

IntBufferSwap example revisited

- See folder 4.
- Focus on iterator and const_iterator.

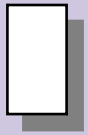
Iterators & Sequence Containers



```
SeqContainerName<...> c;
SeqContainerName<...>::iterator i,j;
```

- `c.insert(i,x)` – inserts `x` before `i`
- `c.insert(i,first,last)`
  – inserts elements in `[first,last)` before `i`
- `c.erase(i)` – erases the element that `i` points to
- `c.erase(i,j)`
  – erase elements in range `[i,j)`

Iterators & Sequence Containers c++11



```
SeqContainerName<...> c;
SeqContainerName<...>::iterator i,j;
```

- `c.emplace(i,p1,...,pn)`:

Constructs and inserts before `i` an object with a constructor that gets `p1,...,pn` parameters

Iterators & other Containers

- insert and erase has the same ideas, except they keep the invariants of the specific container.
- For example, a Sorted Associative Container will remain sorted after insertions and erases.

Iterators & other Containers

- So what does `c.insert(pos,x)` does, when c is a Unique Sorted Associative Container ?

- Inserts x into the set, using **pos** as a **hint** to where it will be inserted.

Iterators & other Containers: c++11

- So what does `c.emplace_hint(pos,x)` does, when c is a Unique Sorted Associative Container?

- Constructs and Inserts x into the set, using **pos** as a **hint** to where it will be inserted.

Iterator validity

- When working with iterators, we have to remember that their validity can change

What is wrong with this code?

```
Container<...> c;
...
for(auto i= c.begin(); i!=c.end(); ++i )
    if( f( *i ) ) { // some test
      c.erase(i);
    }
```

Iterator validity

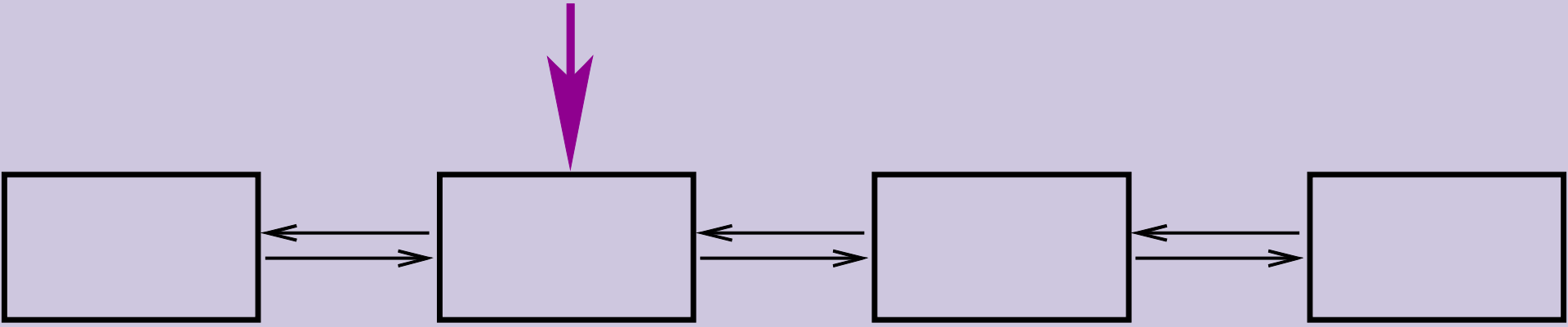
Two cases:

- list, set, map
  - `i` is not a legal iterator

Iterator validity

Two cases:

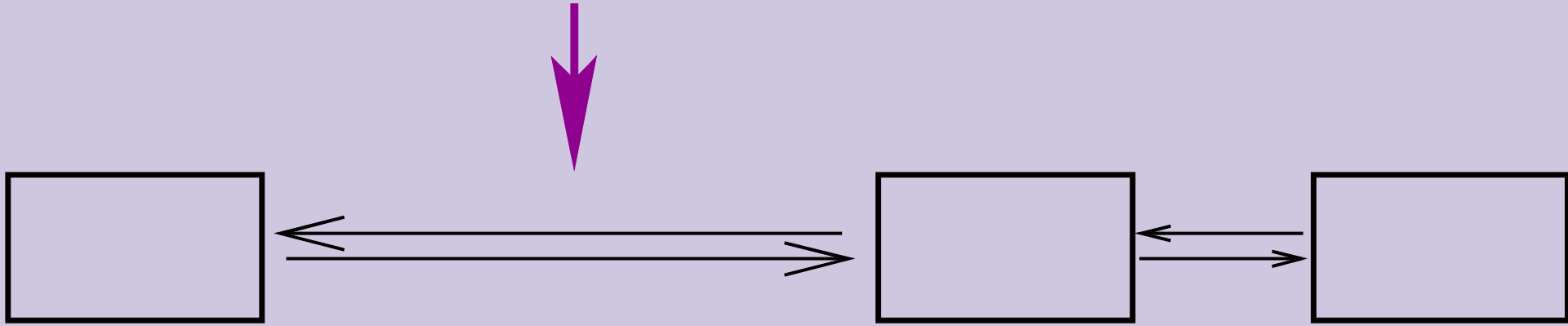
- **list**, set, map
  - `i` is not a legal iterator



Iterator validity

Two cases:

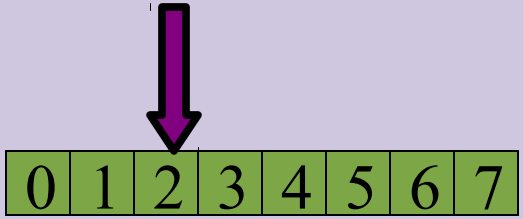
- **list**, set, map
  - `i` is not a legal iterator



Iterator validity

Two cases:

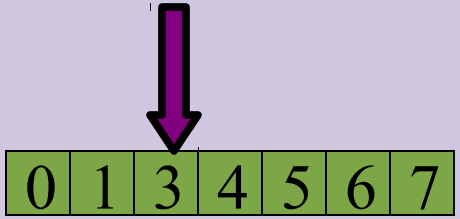
- list, set, map
  - `i` is not a legal iterator
- **vector**
  - `i` points to the element after



Iterator validity

Two cases:

- list, set, map
  - `i` is not a legal iterator
- **vector**
  - `i` points to the element after



Iterator validity

Two cases:

- list, set, map
  - `i` is not a legal iterator
- vector
  - `i` points to the element after

In either case,

this is not what we want…

Erasing during iteration (folder 5)

```
Container<...> c;
...
for(auto i= c.begin(); i!=c.end();/*no ++i*/ )
  if( f( *i ) ) { // some test
    i = c.erase(i);
  } else {
    ++i;
  }
```

Iterators & Map

Suppose we work with:

```
map<string,int> dictionary;
map<string,int>::iterator it;
…
it = dictionary.begin();
```

What is the type of `*it` ?

Iterators & Map

Every STL container type Container defines

`Container::value_type`

Type of elements stored in container

- This is the type returned by an iterator

```
Container::value_type operator*();
```

Iterators & Map

- Ok, so what type of elements does a map return?

- **`map<KeyType,ValueType>`** keeps pairs
  - **`KeyType key`** – “key” of entry
  - **`ValueType value`** – “value” of entry

Pairs

```
template< typename T1, typename T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;

  pair( const T1& x, const T2& y )
     : first(x), second(y)
  {}
};
```

Map value_type

```
template< typename Key, typename T,
          typename Cmp = less<Key> >
class map {
public:
  typedef pair<const Key, T> value_type;

  typedef Key key_type;
  typedef T mapped_type;
  typedef Cmp key_compare;
};
```

Using map iterator

```
map<string,int> dict;
…

for( auto i = dict.cbegin();
     i != dict.cend();
     ++i )
{
   cout << i->first << " "
        << i->second << "\n";
}
```

Using map iterator

```
  map<string,int> dict;
  …

  for( const auto& val : dict) {
     cout << val.first << " "
          << val.second << "\n";
}
```

Iterators and Assoc. Containers (folder 6)

Additional set of operations:

- **`iterator C::find(key_type const& key)`**

Return iterator to first element with **`key`**.

Return **`end()`** if not found

- **`iterator C::lower_bound(key_type const& key)`**

Return iterator to first element greater or equal to **`key`**

- **`iterator C::upper_bound(key_type const& key)`**

Return iterator to first element greater than **`key`**

Adaptors

- Good functionality, wrong interface
- For example, adaptors of basic containers

with limited interface:

stack<T,SequentialContainer>

queue<T,SequentialContainer>

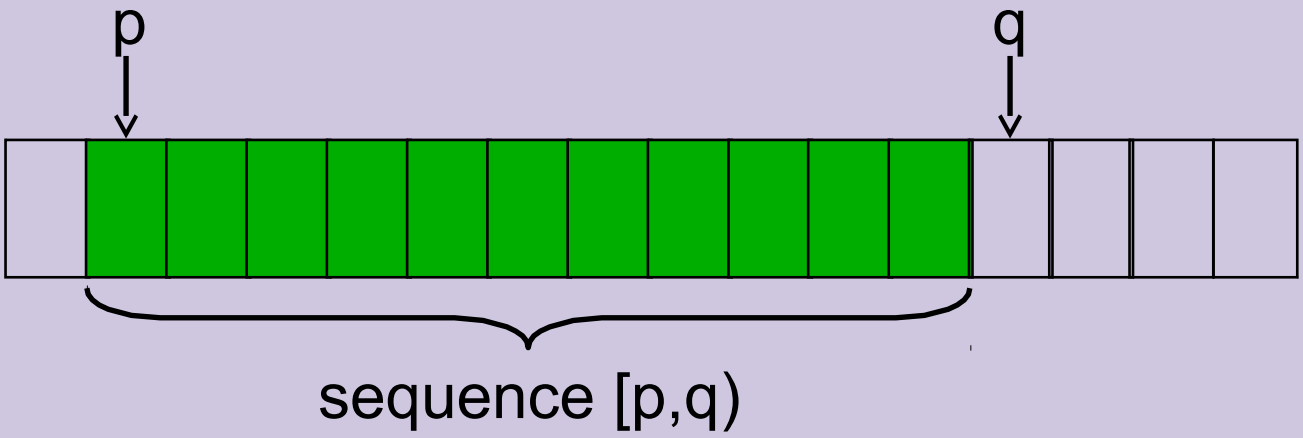
stack<T,SequentialContainer**>**

- provides **`emplace, push, pop, top, size, empty,...`**
- Notice that unlike java, **`pop`** , is not returning a value. i.e: it's a void function.
- The reason (historic with c++-11?):

  to make pop return a value it would be either inefficient or wrong: http://www.sgi.com/tech/stl/stack.html#3

Algorithms

- Most STL algorithms works on sequences
- Sequences are passed as two iterators:
  - beginning element
  - element one after last



- Algorithms depend on iterator type **not** on container type

Example – merge documentation

copy

```
template< typename In, typename Out>
Out copy(In first, In last, Out res)
{
  while (first != last)
    *res++ = *first++;
  return res;
}
```

copy

```
template< typename In, typename Out>
Out copy(In first, In last, Out res)
{
  while (first != last)
    *res++ = *first++;
  return res;
}
```

What's wrong with this ?

```
void foo(const vector<char>& v) {
   vector<char> v2;
   ...
   copy(v.begin(),v.end(), v2.begin());
```

copy

```
template< typename In, typename Out>
Out copy(In first, In last, Out res)
{
  while (first != last)
    *res++ = *first++;
  return res;
}
```

What's wrong with this ?


```
void foo(const vector<char>& v) {
   vector<char> v2;
   ...
   copy(v.begin(),v.end(), v2.begin());
```

So how can we copy and insert ?

Solution #1: Use insert explicitly

```
void foo(const vector<char>& v) {
   vector<char> v2;
   ...
   v2.insert(v2.end(),v.begin(),v.end());
```

So how can we copy and insert ?

Solution #2: Use back_inserter, which returns an iterator that knows to “push_back”. See folder 6.

```
void foo(const vector<char>& v) {
   vector<char> v2;
   ...
   copy(v.begin(),v.end(), back_inserter(v2));
```

sort – using operator <

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Example usage(the hard way):

```
sort< vector<int>::iterator >
                    (vec.begin(), vec.end());
```

sort – using operator <

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Example usage:

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

sort – using operator <

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Example usage with primitive arrays:

```
int arr[5];
...
sort(arr, arr+5);
```

sort – using operator <

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Example usage with primitive arrays (C++11):

```
int arr[5];
...
sort(begin(arr), end(arr));
```

sort – using comparator

```
template <class RandomAccessIterator,
          class StrictWeakOrdering>
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Example usage:

```
sort(vec.begin(), vec.end(), greater<int>());
```

sort – compile error

```
list<int> l(nums, nums+SIZE);
sort(l.begin(), l.end());
```

sort – compile error

```
list<int> l(nums, nums+SIZE);
sort(l.begin(), l.end());
```

list iterators are bidirectional and not random access !

g++

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h: In function ‘void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]’:

Main.cpp:17: instantiated from here

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2713: error: no match for ‘operator-’ in ‘__last - __first’

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iterator_base&, const std::_Bit_iterator_base&)

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h: In function ‘void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]’:

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2714: instantiated from ‘void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]’

Main.cpp:17: instantiated from here

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2357: error: no match for ‘operator-’ in ‘__last - __first’

. . .

g++

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h: In function ‘void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]’:

Main.cpp:17: instantiated from here

/usr/lib/gcc/i486-linux-g... 3: error: no match for ‘operator-’ in ‘__last

/usr/lib/gcc/i486-linux-g... 82: note: candidates are: ptrdiff_t std::operato... ator_base&)

/usr/lib/gcc/i486-linux-g... unction ‘void std::__final_insertior... ator) [with _RandomAccessIter...

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2714: instantiated from ‘void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]’

Main.cpp:17: instantiated from here

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2357: error: no match for ‘operator-’ in ‘__last - __first’

. . .

g++

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h: In function ‘void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with **_RandomAccessIterator = std::_List_iterator<int>**]’:

Main.cpp:17: instantiated from here

/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/stl_algo.h:2713: error: no match for ‘operator-’ in ‘__last - __first’

...

g++ -D_GLIBCXX_CONCEPT_CHECKS and STLFilt

```
BD Software STL Message Decryptor v2.47a for gcc
stl_algo.h: In function ‘void sort(_List_iterator<int>,
_List_iterator<
          int>)’:
Main.cpp:17:   instantiated from here
stl_algo.h:2713: error: no match for ‘operator-’ in ‘__last - __first’
stl_algo.h: In function ‘void __final_insertion_sort(
       _List_iterator<int>, _List_iterator<int>)’:
stl_algo.h:2714:   instantiated from ‘void sort(
       _List_iterator<int>, _List_iterator<int>)’
Main.cpp:17:   instantiated from here
```

• • •

g++ -D_GLIBCXX_CONCEPT_CHECKS and STLFilt

…
Main.cpp:17: instantiated from here
boost_concept_check.h:223: error: **conversion from ‘**
bidirectional_iterator_tag’ to non-scalar type ‘
random_access_iterator_tag’ requested

Cryptic error messages

STLFilt:

An STL Error Message Decryptor for C++:

http://www.bdsoft.com/tools/stlfilt.html

Cryptic error messages

Different compilers:

- clang++ (free)
- intel c++ (not free)

Strings

What is a string ?

- a typedef for basic_string<char>
- The basic_string class represents a Sequence of characters.
- It contains:
  - all the usual operations of a Sequence.
  - standard string operations such as search and concatenation.

Rope: non standard!

- Scalable string implementation.
- Efficient operation that involve the string as a whole.
- A rope is a reasonable representation for very long strings such as edit buffers or mail messages.

Rope: non standard!

- Scalable string implementation.
- Efficient operation that involve the string as a whole.
- A rope is a reasonable representation for very long strings such as edit buffers or mail messages.

How to convert something to a string?

- c++11: to_string for primitives

- Using std::ostringstream
- We can encapsulate the string conversion and format into stringify functions – **stringify example**

More?

- Lots of other features, especially in c++11 (threads,…)

- Other libraries:
  - Boost
  - opencv, dlib, armadillo, zlib, …