The Standard Template Library

Containers,
algorithms
and more...

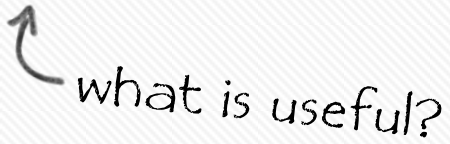
The Standard Library

- The **standard library** is a collection of **functions** and **data types** that is supplied with every **standard-compliant C++ compiler**

- What **should** be in the standard library?
  - And equally important: what **should not** be?

The Standard Library

- The standard library should:
  - Provide **basic functionality** which the user can have difficulty implementing himself:
    - Memory management
    - Input / output
    - Algorithms, Containers …
  - Reduce and **simplify work**:
    - Supply useful non-primitive facilities

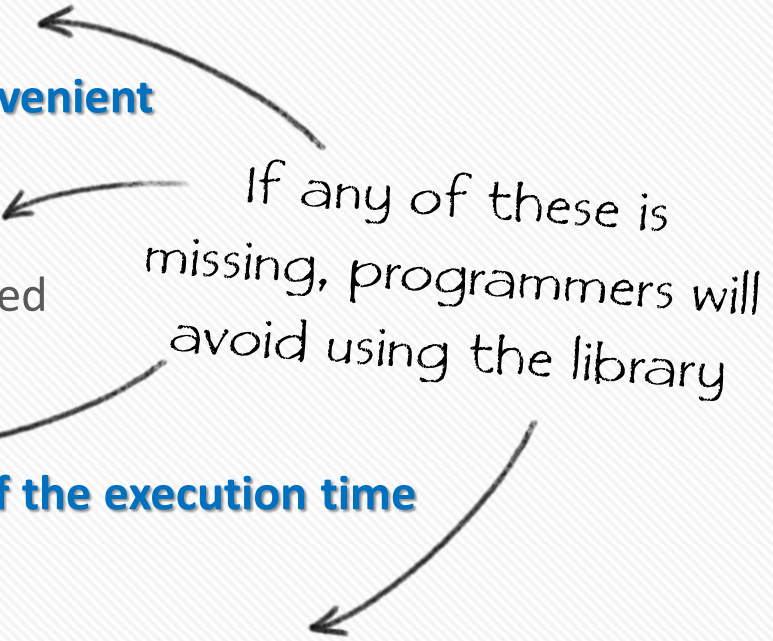


The Standard Library

- The standard library cannot contain **problem-specific code**
  - For example – a Student class
- It should contain generic classes and functions which are **usable in a variety of problems**:
  - **Containers** – such as list, set, etc.
  - **Algorithms** – such as sorting, searching
  - **Popular classes** – such as string

The Standard Library

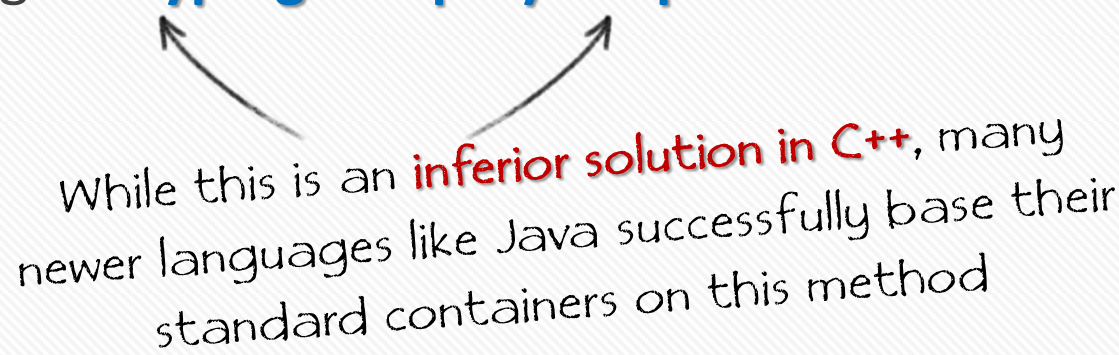
- What are the **requirements** from the standard library's containers and algorithms?
  - **Simple** to use
    - Interfaces should be **uniform and convenient**
  - **Complete**
    - **Reasonable** features should be included
  - **Efficient**
    - Library code might take a **large part of the execution time**
  - **Extensible**
    - The user should be able to **add new containers and algorithms** that work with the given ones



Designing Containers

- The containers included in the standard library should be **generic**

- We have seen several ways to implement generic containers:
  - Using **void* and pointers to functions**
  - Using **templates**
  - Using **subtyping** and **polymorphism**

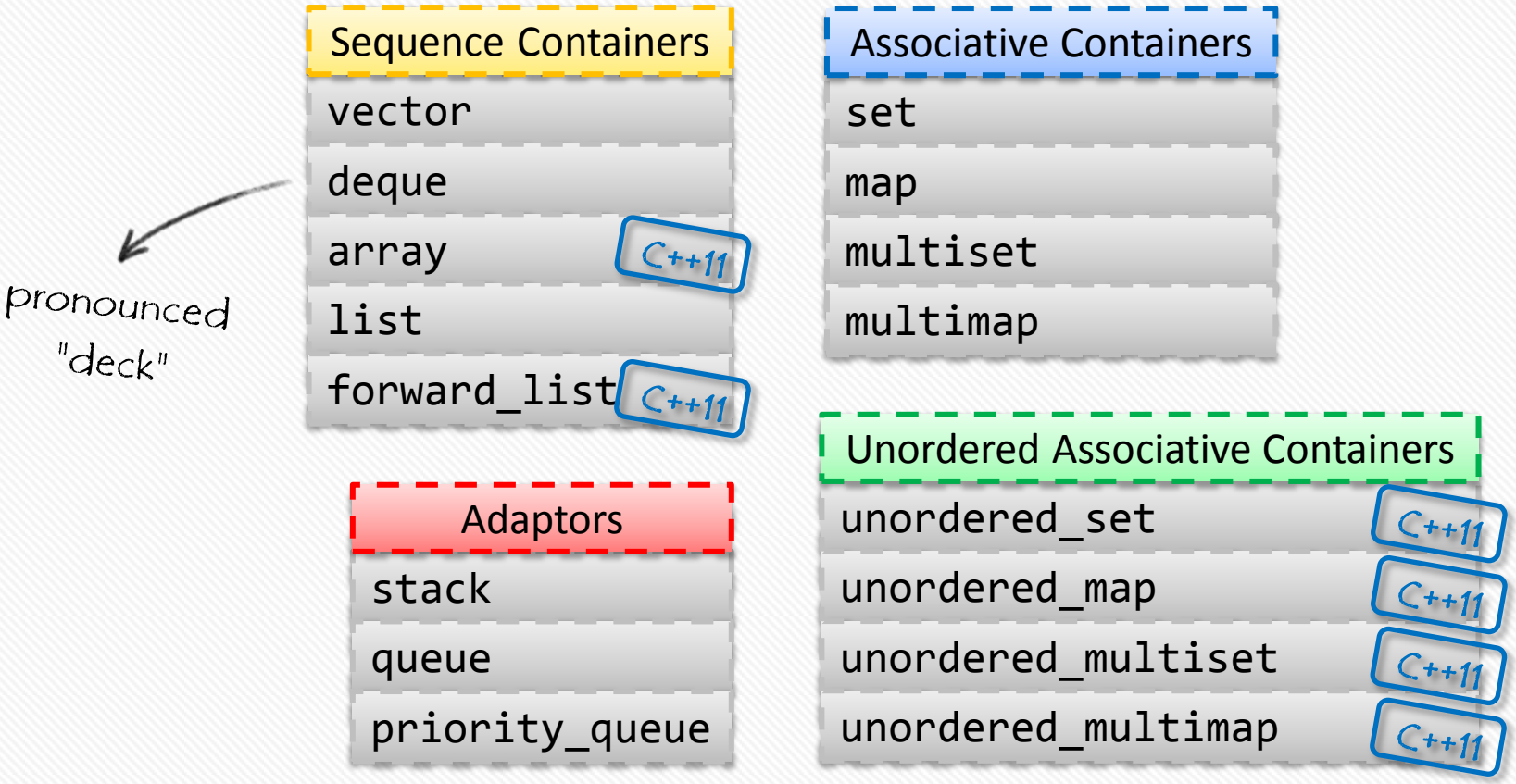


The Standard Template Library

- The **standard template library** (**STL**) is a collection of **containers** and **algorithms** which is part of the standard C++ library
- Makes heavy use of **C++ templates**
  - And many interesting and advanced programming techniques
- Developed by Alexander Stepanov and Meng Lee at HP

Containers

- The STL includes several generic containers:



Vector

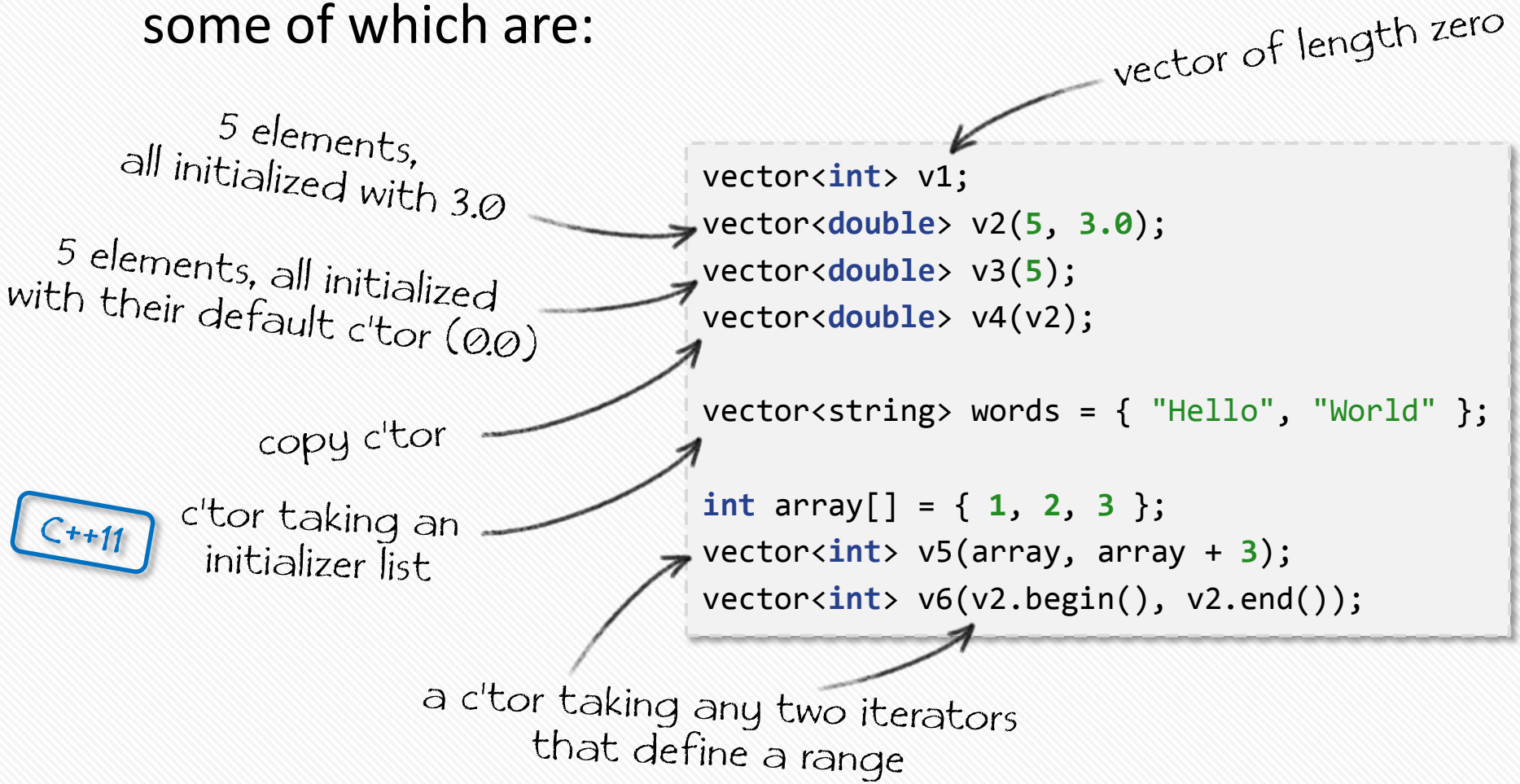
- The most **commonly used** container is **std::vector**
  - Defined in the standard header <vector>
- Essentially, a vector is a **dynamic array**
  - Elements are **sequential in memory**
  - **Fast random access** to elements by index
  - Insertion and removal of elements at the **end of the vector**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

```
void read_and_print() {
  vector<int> numbers;
  int input;
  while (cin >> input) {
    numbers.push_back(input);
  }
  for (int i = 0; i < numbers.size(); ++i) {
    cout << numbers[i] << " ";
  }
}
```

append an element to the end of the vector

Initializing Vectors

- std::vector has a large selection of **constructors**, some of which are:



Accessing Vectors

- Vectors have **random access** just like C arrays:
  - **operator[]** can be used to access elements (the result is **undefined** if the index is invalid, just like C arrays!)
  - The member function **at()** provides a safer (but slower) version of operator[], which **throws an exception** if the index is illegal

prefer at() when accessing a vector not within a safe loop. Only switch to [] if efficiency becomes an issue

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

for (int i = 0; i < words.size(); ++i) {
  cout << words[i] << endl;
}

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

Vector Iteration

- std::vector supports **iteration**
  - **begin() and end()** provide iterators to the first and one-after-last element of the vector

```
vector<int> v = { 2, 3, 4, 1, 2, 4 }
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
  }
}
```

2 3 4 1 2 4

  - **rbegin() and rend()** provide iteration in reverse order

```
vector<int> v = { 2, 3, 4, 1, 2, 4 }
for (vector<int>::iterator it = v.rbegin(); it != v.rend(); ++it) {
    cout << *it << " ";
  }
}
```

4 2 1 4 3 2

C++11 Foreach Loop

- The following loop structure is very common in C++:

```
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
  int num = *it;
  // ... do something with num
}
```

- C++11 adds built-in **for each** loops:

```
for (int num : vec) {
  // ... do something with num
}
```

C++11

equivalent

- This syntax works for:
  - All standard C++ containers
  - Normal C-style arrays (not pointers to arrays though)
  - Any user-defined class that has begin() and end() functions, and which return a **valid iterator object** (i.e., an object with the three basic iterator operations *, ++, !=)

Insertion and Removal

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

{ "Hello", "World", "!!!" }

back to the original content

Allocate in advance 10 items

There are actual 5 items

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

words.push_back("!!!");

words.pop_back();

words.reserve(10);
words.resize(5);
```

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

inserts element before the iterator
(++it points to "World")
{ "Hello", "Lovely", "World" }

erases "Hello", because the iterator refers to it

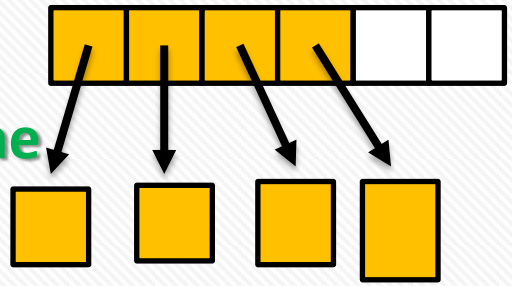
```
vector<string> words = { "Hello", "World" };

vector<string>::iterator it = words.begin();
words.insert(++it, "Lovely");

words.erase(words.begin());
```

Vector - Notes

- Elements placed into a vector must be "**copy constructible**" and "**assignable**"
  - So a working **copy c'tor** and **operator=** are required
  - A default c'tor is **not required**

unlike ordinary arrays

- When storing pointers to dynamic objects, use **shared_ptr** to store them in the vector
  - This will spare you from the need to explicitly delete elements before removing them from the vector

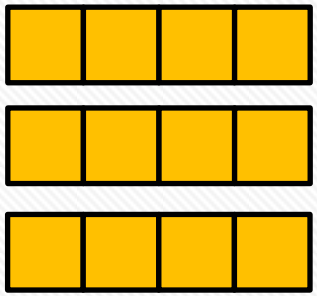


- There is more to vector's interface – **look online**
  - This is true for **all containers**

Other Sequence Containers

- The rest of the containers behave similarly to vector

- The container **std::deque** (short for "Double Ended Queue") is like a vector, but allows fast insertion and removal from **both its sides**
  - Using the push_front() and pop_front() methods



{ "Hi", "World" }

```
deque<string> words = { "Hello", "World" };
words.pop_front();
words.push_front("Hi");
```

Other Sequence Containers

- The container **std::list** is implemented as a doubly-linked list
  - There are no methods for index-based access
  - There is support for fast insertion, deletion, merging, and splicing

move nodes from one list to another

```
list<string> words = { "Hello", "World" };
list<string> words2 = { "Lovely" };
words.splice(++words.begin(), words2);
```

{ "Hello", "Lovely", "World" }

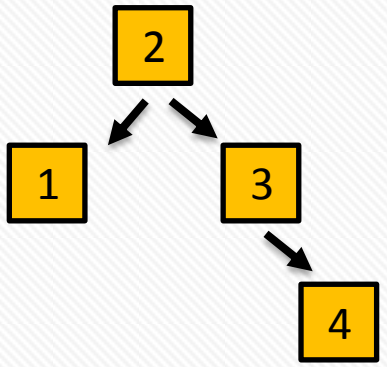
words2 is now empty

Associative Containers

- Unlike sequence containers, associative containers **do not provide control over element order**
  - The order of iteration is unrelated to the order of insertion
- Example: **set** is an associative container

```
vector<int> values = { 1, 3, 2, 4, 3 };

set<int> numbers;
for (int n : values) {
  numbers.insert(n);
}

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

1 2 3 4



Associative Containers

- **set** stores its elements in sorted order
  - By default, set compares elements using operator<()
  - When **using pointers**, this is usually bad

```
class Employee {
  string name;
public:
  Employee(string name)
    : name(name) {}
  string getName() const {
    return name;
  }
  // ...
};
```

```
set<Employee*> employees;

employees.insert(new Employee("John"));
employees.insert(new Employee("Jill"));
employees.insert(new Employee("John"));

for (const Employee* e : employees) {
  cout << e->getName() << endl;
}
```

this set will contain 3 (not 2!) employees, and their order will depend on their address in memory

Associative Containers

- A comparison function can be passed to **set** as a template argument
  - We can replace the default operator<() with a **function object**

```
class CompareByName {
public:
  bool operator()(const Employee* e1, const Employee* e2) const
  {
    return e1->getName() < e2->getName();
  }
};
```

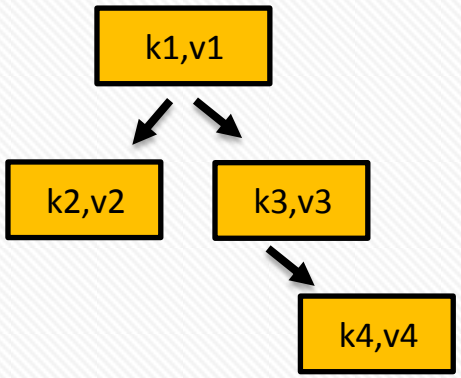
```
class Employee {
  string name;
public:
  Employee(string name)
    : name(name) {}
  string getName() const {
    return name;
  }
  // ...
};
```

```
set<Employee*, CompareByName> employees;

employees.insert(new Employee("John"));
employees.insert(new Employee("Jill"));
employees.insert(new Employee("John"));

for (const Employee* e : employees) {
  cout << e->getName() << endl;
}
```

now these two are equal and will only be added once

*Remember: prefer using shared_ptr to simple pointers

Associative Containers



- Another associative container is **map**
  - A map stores pairs of **(key, value)**
  - The **keys are unique** (only one of each)
  - Map is basically a set, with extra information stored for each element

```
map<int, shared_ptr<Employee>> idToEmp;
idToEmp[123456789] = shared_ptr<Employee>(new Employee("John"));
idToEmp[987654321] = shared_ptr<Employee>(new Employee("Jill"));
idToEmp[123454321] = shared_ptr<Employee>(new Employee("John"));

idToEmp[123456789] = shared_ptr<Employee>(new Employee("Jack"));

for (const pair<int, shared_ptr<Employee>>& idAndEmp : idToEmp) {
  cout << idAndEmp.first << ": " << idAndEmp.second << endl;
}
```

changes the Employee associated with this id

key (id number)

value (employee)

Another Map Example

for each word, count its number of appearances

find the (key, value) pair with the largest value

```
void most_frequent_word() {
  string word;
  map<string, int> count;
  while (cin >> word) {
    count[word]++;
  }

  string max_word;
  int max_count = 0;
  for (const pair<string, int>& p : count) {
    if (p.second > max_count) {
      max_word = p.first;
      max_count = p.second;
    }
  }

  cout << max_word << endl;
}
```

repeat until EOF

the first time a key is accessed, its value is set to 0 (default c'tor of int)

Containers - Final Notes

- Don’t feel confused by the large number of different containers
  - If you don't know which to use, **go with vector**
  - In time, the **differences will become clearer**
- How each of these containers can be implemented efficiently is taught in *Data Structures 1 (234218)*

Iterators

- A common feature of most containers is **iterators**
  - All with **the same interface**
- STL containers return iterators to their elements via **begin()** and **end()**
- The iterators supports (at least) the **3 basic operations**:
  - **Advancement** (++) , **dereferencing** (*, ->), and **comparison** (==, !=)
  - Some may support additional operations, such as **move backwards** (--) (std::list), or **direct access** (operator[]) (std::vector, std::deque)

```
list<int> numbers = { 1, 2, 3, 4 };
list<int>::iterator i = numbers.begin();
while (i != numbers.end()) {
  *i = 2 * (*i);
  ++i;
}
```

advancing

dereferencing

comparison

or in C++11 for-each loop syntax...

```
for (int& num : numbers) {
  num = 2 * num;
}
```

Iterators

- Having a **consistent interface** allows us to write algorithms that can work with **multiple containers**

```
template<class Iterator>
void printElements(Iterator first, Iterator last) {
  while (first != last)
    cout << *(first++);
}
```

```
list<int> list;
vector<int> vec;
// ...
printElements(list.begin(), list.end());
printElements(vec.begin(), vec.end());
```

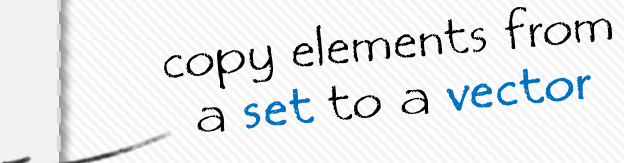
use the same function with a list and a vector!

Iterators

- Having a **consistent interface** allows us to write algorithms that can work with **multiple containers**

```
template<class InputIter, class OutputIter>
void copyElements(InputIter first, InputIter last, OutputIter target) {
  while (first != last) {
    *(target++) = *(first++);
  }
}
```

```
set<int> values;
// ...
vector<int> vec(values.size());
copyElements(values.begin(), values.end(), vec.begin());
```



Iterators

- Having a **consistent interface** allows us to write algorithms that can work with **multiple containers**

```
template <class InputIter1, class InputIter2, class OutputIter>
void merge(InputIter1 first1, InputIter1 last1,
           InputIter2 first2, InputIter2 last2,
           OutputIter target)
{
  while ((first1 != last1) && (first2 != last2)) {
    if (*first1 < *first2) {
      *(target++) = *(first1++);
    }
    else {
      *(target++) = *(first2++);
    }
  }
  copyElements(first1, last1, target);
  copyElements(first2, last2, target);
}
```

merge sorted sequences

destination must have space for the result before calling merge()

```
list<int> list1 = { 1, 3, 5, 7, 9 };
list<int> list2 = { 2, 4, 6, 8, 10 };
vector<int> vec(list1.size() + list2.size());

merge(list1.begin(), list1.end(),
      list2.begin(), list2.end(),
      vec.begin());
```

Algorithms

- STL comes with built-in implementations for several **algorithms**
  - All operate on **iterators**

```
#include <algorithm>
```

```
template<class Iterator, class T>
int std::count(Iterator first, Iterator last, const T& value);
```

```
vector<int> numbers = { 1, 5, 2, 5, 7 };
int counter = count(numbers.begin(), numbers.end(), 5);   // returns 2
```

- Some algorithms require iterators with **particular operations**, such as an operator[]

```
template<class Iterator>
void std::sort(Iterator first, Iterator last);
```

← requires an operator[]

```
vector<int> my_vec = { 3, 6, 2, 1, 5 };
sort(my_vec.begin(), my_vec.end());
list<int> my_list  = { 3, 6, 2, 1, 5 };
sort(my_list.begin(), my_list.end());
```

← O(nlog(n))

← does not compile...

Algorithms

- Other algorithms automatically use the **most efficient implementation** based on the operations the input iterator supports

```
template<class Iterator, class T>
bool std::binary_search(Iterator first, Iterator last, const T& value);
```

```
list<int> my_list  = { 1, 3, 4, 5, 9 };
vector<int> my_vec = { 1, 3, 4, 5, 9 };
bool result1 = binary_search(my_list.begin(), my_list.end(), 6);
bool result2 = binary_search(my_vec.begin(), my_vec.end(), 6);
```

O(n)

O(log(n))

- There are many more algorithms!
  - std::find() – find the first occurrence of an element
  - std::count_if() – count the elements that satisfy a condition
  - std::reverse() – reverse the order of elements
  - std::min(), std::max() – return the minimum/maximum value in range

Summary

- The STL supplies a set of useful tools for programming in C++
  - Use it whenever possible instead of **reinventing the wheel**
- Don’t expect to remember it by heart – it’s too big
  - **Look online** for resources, often what you need already exists
  - There's much more to it, but outside the scope of this course
  - An excellent series of lectures (by one of the writers of STL): https://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-/C9-Lectures-Introduction-to-STL-with-Stephan-T-Lavavej
- The STL offers many containers, but in most cases there is no need to optimize anything
  - If you're not sure what you need – **use a vector**

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

- Martin Fowler