DigiPen
INSTITUTE OF TECHNOLOGY

CS170#03.2

Introduction To STL And Vectors

Vadim Surov

Outline

- STL
- Vector
- Iterators
- Algorithms

DigiPen
INSTITUTE OF TECHNOLOGY

Standard Template Library

STL

- Standard Template Library – general purpose library of generic algorithms and data structures (containers)
- It contains a lot of different components - the most commonly used containers and algorithms in computer science
  - An ISO C++ standard framework of about 10 containers and about 60 algorithms
- Containers and algorithms are independent
- Algorithms use iterators to get access to data

Why STL?

- It is thoroughly tested
- Portable code
- Highly efficient
  - STL data structures grow automatically
  - May contain machine-based optimisations

So:

- If you can find it in the STL, it is usually better to use it than to write your own!
- Understanding STL is important to improve your productivity

STL Components: Containers

- E.g.: `vector`, `list`, `set`, `stack`, `queue`, `map`
- Implemented as templates (type parameterized)
- Stores and manages a collection of objects of the same type according to a specific organization
- Different containers are suitable for different purposes

STL Components: Iterators

- Used to traverse ("walk") a container
- Provide a common interface for iterating over the elements in a container
- Similar to pointers in that an iterator will "point" to the "current" object in a container
- What is the type of a "current" object varies depending on the container

STL Components: Algorithms

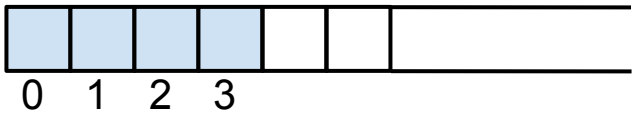
- E.g.: `count()`, `copy()`, `sort()`, `search()`
- Applied to (ranges within) containers to process the elements
- Algorithms work closely with iterators
- They are often referred to as **generic algorithms** because they can be applied to almost any container (that contains almost anything)

DigiPen
INSTITUTE OF TECHNOLOGY

Vector

The vector Container

- One of the most popular STL containers is the `vector`
- It is a template class
- It implements a dynamic array with added capabilities
- You need to **`#include`** `<vector>` to use it



The vector Container (contd)

- Basic usage:

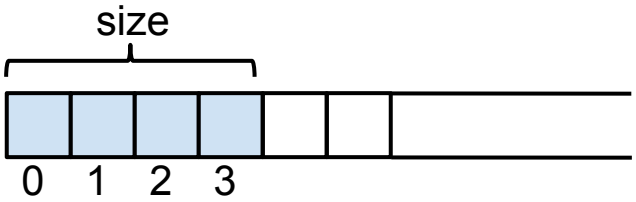
```
#include <vector>

void foo() {
  const int SIZE = 5;
  std::vector<int> numbers(SIZE); // vector of 5 int
  for (int i = 0; i < SIZE; i++) // assign values
    numbers[i] = i;

  numbers.resize(numbers.size() * 2); // double the size
  for (int i = SIZE; i < SIZE * 2; i++) // assign more values
    numbers.at(i) = i;
}
```

The vector Container (contd)

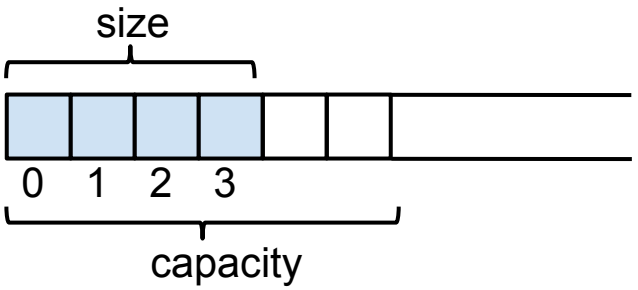
- The **size** of a vector (`size()`) is the number of actual elements that it is holding



- The default constructor is called for all elements
- Using subscript operator `[]` to access elements beyond the size causes undefined behaviour
- Using the `at()` method to access elements beyond the size throws an exception
- Use `resize()` to increase or decrease the size

The vector Container (contd)

- The **capacity** of a vector (`capacity()`) is the maximum number of elements it can contain before reallocation is needed



- You can use `reserve()` to change the capacity of a vector
  - it changes capacity to some value >= to the argument
- The `push_back()` method adds an element to the back of the vector (automatically resizing if necessary)

The vector Container (contd)

```
void foo() {
  std::vector<int> numbers; // create an empty vector of int
  for (int i = 0; i < 5; i++) // assign values
    numbers.push_back(i);
    // print values
  for (size_t i = 0; i < numbers.size(); i++)
    cout << numbers[i] << " ";
  cout << endl;
    // Show size and capacity
  cout << "Size: " << numbers.size() << endl;
  cout << "Capacity: " << numbers.capacity() << endl;
    // Remove all elements (calls destructors)
  numbers.clear();
    // Show size and capacity
  cout << "Size: " << numbers.size() << endl;
  cout << "Capacity: " << numbers.capacity() << endl;
}
```

```
0 1 2 3 4
Size: 5
Capacity: 8
Size: 0
Capacity: 8
```

The vector Container (contd)

- Often used methods for vector:
  - `push_back()` - add an element to the end; (constant time unless capacity reached, whereupon it is linear time)
  - `insert()` - add at a location (linear time)
  - `begin()` - returns iterator to start of the vector (constant time)
  - `end()` - returns iterator to one past the end of the vector, respectively (constant time)
  - `erase()` - deletes an element or range of elements, capacity is no reduced (linear time)
  - `clear()` - deletes all elements, capacity is not reduced (linear time)

The vector Container (contd)

- Often used methods for vector (contd):
  - `empty()` - returns true if empty, false otherwise (constant time)
  - `operator[]`, `at()` - access an element by index; (constant time)
  - `front()`, `back()` - access first/last element respectively (constant time)
  - `size()` - returns number of elements (constant time)
  - `capacity()` - returns number of elements that can be contained without reallocation (constant time)
  - `swap()` - exchanges all elements in two vectors (constant time)

DigiPen
INSTITUTE OF TECHNOLOGY

Iterators

Iterators

- Iterators are a generalization of pointers that allow a programmer to work with different containers in a uniform manner
- All of the standard containers (except adaptors) support iterators, so the code for accessing elements is similar between containers
- Iterators are objects so they have an interface that clients can use
  - It hides the implementation details of the container
    - Ex: code for traversing an array is different than code for traversing a linked-list
- Since iterators are a generalization of pointers it is assumed that every template function that takes iterators as arguments also works with regular pointers

An Example Of Using Iterators On A Vector

```
 // Create vector, add 5 integers
vector<int> cont1;
for (int i=0; i<5; ++i)
  cont1.push_back(i);

 // Create an iterator of the proper type
vector<int>::iterator iter;

 // Iterate over the container to
 //   print each element
for (iter=cont1.begin(); iter!=cont1.end(); ++iter)
  std::cout << *iter;
```

```
0 1 2 3 4
```

Iterator Categories

- There are several categories of iterators, each providing different capabilities (to satisfy algorithm's requirements):
  - **Forward** – can read, write, move in one only direction (forward) from one element to the next
  - **Bidirectional** – can read, write, move forward and backward from one element to the next
  - **Random Access** – can read, write, access any element at any time (from any other element)

- Note: Not all containers provide all types of iterators.
  Ex: list doesn't provide random access

Iterator Categories (contd)

- Different containers provide different iterators:

Container	Iterator category
vector, deque, string	random access
list, set, map, multiset, multimap	bidirectional

Forward Iterators

- `iter_type()` – default constructor. Instantiates an iterator of type `iter_type`
- `iter_type(iterator)` – copy constructor. Instantiates an iterator of type `iter_type` from `iterator`
- `*iterator` dereferences the iterator. Returns the object referenced by the iterator
- `iterator->member` returns a member of the object referenced by the iterator

Forward Iterators (contd)

- **++**`iter` increments the iterator (move to the next element). Returns incremented iterator
- `iter`**++** increments the iterator (move to the next element). Returns previous non-incremented iterator
- `iter1` **=** `iter2` assigns `iter2` to `iter1`
- `iter1` **==** `iter2` checks for equality
- `iter1` **!=** `iter2` checks for inequality

Forward Iterator Example

```
vector<int> v(3, 1);
v.push_back(7); // vector v: 1 1 1 7
vector<int>::iterator i =
    find(v.begin(), v.end(), 7);

if (i != v.end())
  cout << *i;
else
  cout << "not found";
```

7

Bidirectional Iterators

- Have all of the capabilities of forward iterators, and add the ability to move backward through the elements:
  - `--iterator` decrements the iterator (move back one). Returns decremented iterator
  - `iterator--` decrements the iterator (move back one). Returns previous non-decremented iterator

Bidirectional Iterator Example

```
list<int> l(1, 1);
l.push_back(2); // list l: 1 2

list<int>::iterator first = l.begin();
list<int>::iterator last = l.end();

while (last != first) {
  --last;
  cout << *last;
}
```

```
2 1
```

Random Access Iterators

- Provide all of the capabilities of bidirectional iterators, but add some other functionality like use of:
  - **subscript operator** `[]` for accessing any element
  - **iterator arithmetic** for moving to any element (much like pointer arithmetic)
    - E.g.: `iter+i; iter-i; iter+=i`
  - **comparison operators** for determining the relative positions of two iterators
    - E.g.: `iter1 < iter2; iter1 > iter2; iter1 <= iter2; iter1 >= iter2;`

Random Access Iterator Example

```
vector<int> v(1, 1); //v: 1
v.push_back(2);      //v: 1 2
v.push_back(3);      //v: 1 2 3
v.push_back(4);      //v: 1 2 3 4
vector<int>::iterator i = v.begin();
vector<int>::iterator j = i + 2;
cout << *j;
i += 3;
cout << *i;
j = i - 1;
cout << *j;
j -= 2;
cout << *j;
cout << v[1] << endl;
```

```
3 4 3 1 2
```

Random Access Iterator Example

```
(j<i) ? cout<<"j<i," : cout<<"not (j<i),";
(j>i) ? cout<<"j>i," : cout<<"not (j>i),";
i = j;
(i<=j && j<=i) ? cout<<"i and j equal" :
                  cout<<"i and j not equal";
cout << endl;
j = v.begin();
i = v.end();
cout << "iterator distance end-begin=size: "
     << (i - j);
```

```
j < i,not (i > j),i and j equal
iterator distance end-begin=size: 4
```

DigiPen
INSTITUTE OF TECHNOLOGY

Algorithms

Generic Algorithms

- The algorithms of the STL are also known as the *generic algorithms* because they are designed and implemented to work with the standard containers
- The capabilities of each algorithm depend on the type of iterator it accepts
- Algorithms behave in different ways. For example some are read-only elements, some modify elements, and some change the order of the elements
- So the generic algorithms can be classified into several categories

Generic Algorithms Categories

1. **Non-modifying -** container will not be changed
2. **Modifying** - change the value of the elements
3. **Removing** - remove some elements from a container
4. **Mutating** - change the order of the elements in a container
5. **Sorting** - mutating algorithms that put the elements of a container in a certain order
6. **Sorted range** - assume that the range to operate on is already sorted
7. **Numeric** - act on numerical elements and combines them based on a specified function

1. Non-modifying Algorithms

- Do not change any elements nor the order of the elements in the container (counting, searching, comparing):
  - `for_each, count, count_if, min_element, max_element, find, find_if, search_n, search, find_end, find_first_of, adjacent_find, equal, mismatch, lexicographical_compare`

2. Modifying Algorithms

- Change the value of the elements:
  - `for_each, copy, copy_backward, transform, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, replace_copy, replace_copy_if`

3. Removing Algorithms

- Removes elements from the container:
  - `remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy`

4. Mutating Algorithms

- Changes the order of the elements in the container:
  - `reverse, reverse_copy, rotate, rotate_copy, next_permutation, prev_permutation, random_shuffle, partition, stable_partition`

5. Sorting Algorithms

- Changes the order of the elements in the container according to a sorting criterion:
  - `sort, stable_sort, partial_sort, partial_sort_copy, nth_element, partition, stable_partition, make_heap, push_heap, pop_heap, sort_heap`

6. Sorted Range Algorithms

- Assumes that the range to operate on is already sorted:
  - `binary_search, includes, lower_bound, upper_bound, equal_range, merge, set_union, set_intersection, set_difference, set_symmetric_difference, inplace_merge`

7. Numeric Algorithms

- Acts on numerical elements and combines them based on a specified function:
  - `accumulate, inner_product, adjacent_difference, partial_sum`

References

- C++ Primer, Fourth Edition
- http://en.wikipedia.org/wiki/Standard_Template_Library
- http://www.cplusplus.com/reference/stl/
- http://www.cplusplus.com/reference/stl/vector/
- http://www.cplusplus.com/reference/iterator/
- http://www.cplusplus.com/reference/algorithm/