STL

Prof. Jinkyu Jeong (jinkyu@skku.edu)
TA -- Minwoo Ahn (minwoo.ahn@csl.skku.edu)
TA -- Donghyun Kim (donghyun.kim@csl.skku.edu)

Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
What is STL?

• Standard Template Library
• Software library for the C++ programming language
• Components of STL
  – Container: Objects that store data
  – Iterator: Generalization of the concept of pointer
  – Algorithms: Procedures that applied to containers
Container

• Objects that store data

• Three types of container
  – Sequence container – vector, list, deque
  – Associative container – set, map, hash_set, hash_map
  – Container adapter – stack, queue, priority queue
Sequence Container – Vector

• Random access is possible like array
• Initially, the capacity of vector is allocated as much as the number of elements
• If there is not enough memory for the new element, allocate larger memory to the vector
  – If you use g++ compiler, new capacity is double of current capacity
  – You can check the capacity of vector through method ‘capacity()’
Sequence Container – Vector

```cpp
vector<int> int_vec = {10, 20, 30};
cout << int_vec.capacity() << endl;
int_vec.push_back(40);
cout << int_vec.capacity() << endl;
```

![Diagram of vector operations](https://via.placeholder.com/150)

- **int_vec**
  - `size()`: current element count
  - `capacity()`: total space available
  - `push_back(40)`: adds 40 to the end
Sequence Container – List

- Doubly linked list
- Elements are not stored in contiguous memory
- Slow lookup and access, but once a position has been found, quick insertion and deletion
Sequence Container – Deque

• Double ended queue
• Vector with insertion/erase at beginning or end
**Associative Container – Set**

- A mathematical set
- Self-balancing binary search tree
- Type of data must implement comparison operator ‘<’ or custom comparator function

![Binary Search Tree Diagram]

```
10 15 20 25 30 35 40
```

SSE2034: System Software Experiment 3, Fall 2018, Jinkyu Jeong (jinkyu@skku.edu)
Operator Overloading

• Customizes the C++ operators for operands of user-defined types

• Operators are treated as a function
  – ex) \(a + b \rightarrow a.\text{operator}+(b)\)

```cpp
return_type operator operator (parameters..)
```

```cpp
class Point{
    int x, y;
    class Point operator+(class Point& p){
        ...
    }
};
```
Associative Container – Map

• An associative array that allows mapping from one data item (key) to another item (value)
• Self-balancing binary search tree
• Type of key must implement comparison operator ‘<’ or custom comparator function
Container Adapter

• Containers with specific interface, using other containers as implementation
  – Stack - vector, deque(default), list
  – Queue - deque(default), list
  – Priority queue - vector(default), deque
Why Iterator is needed?

- Iterator abstracts containers
- Algorithms are implemented with iterator, not direct access of each container
Iterator

- Iterator is similar to pointer
  - Dereference operator (*)
  - Pointer arithmetic

- Each container has its own iterator types
  - Iterator type is dependent on the structure of container

- Five types of iterator
  - Input iterator
  - Output iterator
  - Forward iterator
  - Bidirectional iterator
  - Random access iterator
Method about Iterator

vector<int> v

v.begin()  v.end()

10 20 30 40

deque<int> d

d.begin()  d.end()

block 1

block 2

10

20 30 40 50
### Example

**vector – random access iterator**

```cpp
int main() {
    vector<int> v = {10, 20, 30, 40};
    vector<int>::iterator itr;

    itr = v.begin();
    v.insert(itr + 3, 50); // Random access & pointer arithmetic

    itr = v.begin(); // If an element is added or removed from the container, the iterator is invalidated
    v.erase(itr + 1);

    for(itr = v.begin(); itr != v.end(); itr++) {
        cout << *itr << " ";
    }
    cout << endl;
}
```
Example

list – bidirectional iterator

```cpp
int main() {
    list<int> l = {10, 20, 30, 40};
    list<int>::iterator itr;

    itr = l.end();
    itr--;
    l.insert(itr, 50);

    itr = l.begin();
    itr++;
    l.erase(itr);

    for (itr = l.begin(); itr != l.end(); itr++) {
        cout << *itr << " ";
    }
    cout << endl;
}
```
Algorithm

- A large number of algorithms to perform some activities with iterators
- Each algorithm requires a certain level of iterator
  - sort, stable_sort, partial_sort — random access iterator
  - remove, remove_if — forward iterator
Algorithm – Sort

- \texttt{sort(start, end compare\_func)}
  - Default ordering is ascending order
- \texttt{stable\_sort(start, end, compare\_func)}
- \texttt{partial\_sort(start, middle, end, compare\_func)}

```cpp
#include <algorithm>

struct int_compare {
    bool operator()(const int& a, const int& b) const {
        return a > b;
    }
}

int main() {
    vector<int> v = {10, 40, 20, 30, 5};
    sort(v.begin(), v.end(), int_compare());
}
```
Algorithm – Remove

• remove(start, end, value)

ex) remove(v.begin(), v.end(), 30)

shift except the element that has 30

ex) v.erase(remove(v.begin(), v.end(), 30), v.end())
Algorithm – Remove

• remove_if(start, end, check_func)

```cpp
#include <algorithm>

struct is_odd {
    bool operator()(const int& i) { return i%2 == 1; }
}

int main() {
    vector<int> v = {1, 3, 4, 2, 5, 7, 6};
    v.erase(remove_if(v.begin(), v.end(), is_odd()), v.end());
}
```
Lambda Function

• An unnamed function object capable of capturing variables in scope

```
[captures] (input_params) -> ret {body}
[captures] (input_params) {body}
[captures] {body}
```
Lambda Function Example

```cpp
#include <algorithm>

struct is_odd {
    bool operator()(const int& i) { return i%2 == 1; }
}

int main() {
    vector<int> v = {1, 3, 4, 2, 5, 7, 6};
    v.erase(remove_if(v.begin(), v.end(), is_odd()), v.end());
}
```

```cpp
int main() {
    vector<int> v = {1, 3, 4, 2, 5, 7, 6};
    v.erase(remove_if(v.begin(), v.end(),
                      [](int i) -> bool{return i%2 == 1;}), v.end());
}
```
Lambda Function Example

• Usage of capture list
  - [] : capture nothing
  - [&a, b] : capture ‘a’ by reference, capture ‘b’ by value
  - [&] : capture all variables by reference
  - [=] : capture all variables by value

```cpp
int main() {
    vector<int> v = {1, 3, 4, 2, 5, 7, 6};
    int num_odd = 0;

    v.erase(remove_if(v.begin(), v.end(),
                      [&num_odd](int i) -> bool
                      {num_odd++;
                       return i%2 == 1;}),
            v.end());
}
```
Algorithm – Find

• find(start, end, value)
• find_if(start, end, check_func)

```cpp
#include <algorithm>

int main() {
    vector<int> v = {3, 5, 7, 2, 1};
    auto itr1 = find(v.begin(), v.end(), 3);
    auto itr2 = find_if(v.begin(), v.end(),
                        [](int i){return i%2 == 0;});
}
```
[Lab – Practice #1]

• Make “String” class that works same as standard string class

• Method list
  – Constructor
  – Destructor
  – operator +, +=, ==, >, <
[Lab – Practice #2]

• Input
  – 5 integers or strings

• Procedure
  – Integer: Remove odd numbers using iterator
  – String: Remove strings that start with ‘c’ using iterator

• Output
  – Remain elements

$ ./Lab2
Type: integer
Input: 1 5 2 3 8
Output: 2 8

$ ./Lab2
Type: string
Input: cat cup hat lab cut
Output: hat lab