Standard Template Library (STL)

Modern cpp Programming lecture 6

STL

- We have learned…
  - Basic concepts for the OOP
  - cpp implementations to support the concepts
- However, impractical!!
- Lots of programmers made good tools
  - to support convenient data structure implementation
  - to support specific algorithm (ex) sorting)

STL

- Standard Template Library
  - Software library for the cpp
  - influenced many parts of the C++ Standard Library
  - consists of…
    - Algorithm
    - container
    - functions
    - iterators

STL

- Learning STL?
- Actually, *learning library* is somewhat awkward
- For example…
  - Learning c++ math library??
    - log, exp, sin, cos…
    - only thing to do is to know that ***there exists*** such library
    - Then you can just search for it whenever you want
- Learning STL is same!!

STL

- Learning STL
  - Same as learning c++ math library
  - You only need to know *what* does it contain
  - And I mentioned them already…
    - algorithm
    - container
    - functions
    - iterator
  - You should also remember that these are intertwined each other

STL

- To achieve complete understanding
- You should know
  - Class -> O
  - Overloading -> O
  - Template -> We'll see
  - Function pointer -> We'll see
  - Data structure -> In this lecture
  - Algorithm -> in this lecture
- We haven't learn them…In this lecture, we'll focus on
  - the basic usage & implementation of STL

Data structure & Algorithm

- Simple explanation for Data structure
- Definition
  - organization, management, and storage format that enables efficient access and modification
  - Collection of data values, the relations among them, and the functions of operations that can be applied to the data
- *The efficient ways to store data in computer!!*
- Proper DS selection is important for efficient programming

Data structure & Algorithm

- Algorithm (in CS)
  - Finite sequence of operation
    - manages and control various data structure / variables
    - to solve specific problem
    - quite easy!!
- Combination of DS & algorithm is important!!
  - ex) Array & sorting algorithm

Data structure & Algorithm

- There exists some well-known DS & Algorithm
- Which are empirically proved to be efficient in programming
- became such a **general** programming skills
- STL provides…
  - ***general algorithms for general data structure***
  - *template provides the feature!!*

STL

- Why STL?
  - reduces the cost of programming
  - better maintenance
  - better flexibility & extensibility (by template)
  - High efficiency (already validated)
  - *Standard -> works for every OS & environment*
- *Life is short, use STL!!*

STL

- Simple example
- Suppose that you should implement a list
  - which supports free deletion & insertion of elements

```
int* intArray = new int[100];
for(int i = 0; i < 100; i++) {
   intArray[i] = i;
}
// insertion & deletion??
```

  - Insertion & deletion is quite difficult in this case

STL

```
int* intArray = new int[100];
for(int i = 0; i < 100; i++) {
   intArray[i] = i;
}
// insertion & deletion??
```

- Implementing insertion
  - New allocation? (To store more than 100 elements)
  - Inserting b/w existing elements?
- Implementing deletion
  - handling vacant space?
- These problems are quite tricky to implement

STL

- STL `vector` solves the problems!!

```
vector<int> intVector;
for(int i = 0; i < 100; i++)
   intVector.push_back(i);
```

- Implementing insertion
  - New allocation? (To store more than 100 elements)
    - Automatic memory allocation
  - Inserting b/w existing elements?
    - `insert` method provides the functionality
- Implementing deletion
  - handling vacant space?
    - `delete` method automatically handles those spaces

STL

- Cons
  - Difficult debugging
    - error messages are hard to understand
    - without proper knowledge about cpp & data structures
  - Increases program size
    - Template provides high flexibility but sometimes increases code size
    - As some compilers are not yet optimized

STL

- *Template* in STL

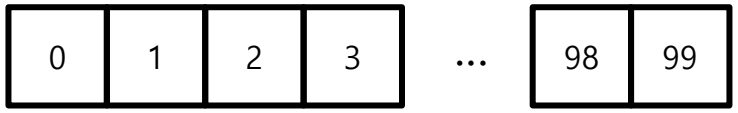
```
vector<int> intVector;
for(int i = 0; i < 100; i++)
   intVector.push_back(i);
```

  - <int> can be replaced to any data structure
    - <string>
    - <int*>
    - <char>
    - or even <vector<int>>
  - offers better extensibility
  - *In next lecture, we'll learn template in detail*

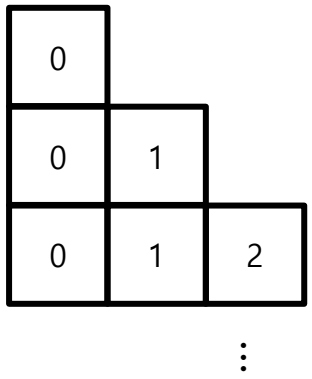
STL example

- vector

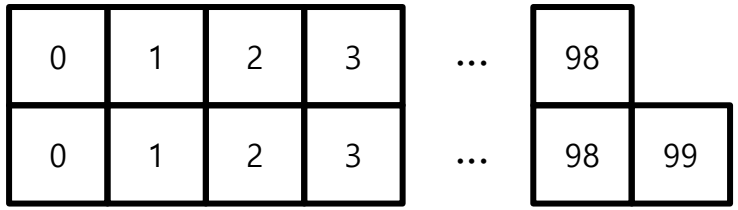
```
vector<int> intVector;
for(int i = 0; i < 100; i++)
   intVector.push_back(i);
```




```
vector<vector<int>> intVector2D;
for(int i = 0; i < 100; i++){
   vector<int> newVector;
   intVector2D.push_back(newVector);
   for(int j = 0; j <= i; j++) {
      intVector2D[i].push_back(j);
   }
}
```



Easy & Flexible 2D array implementation!!



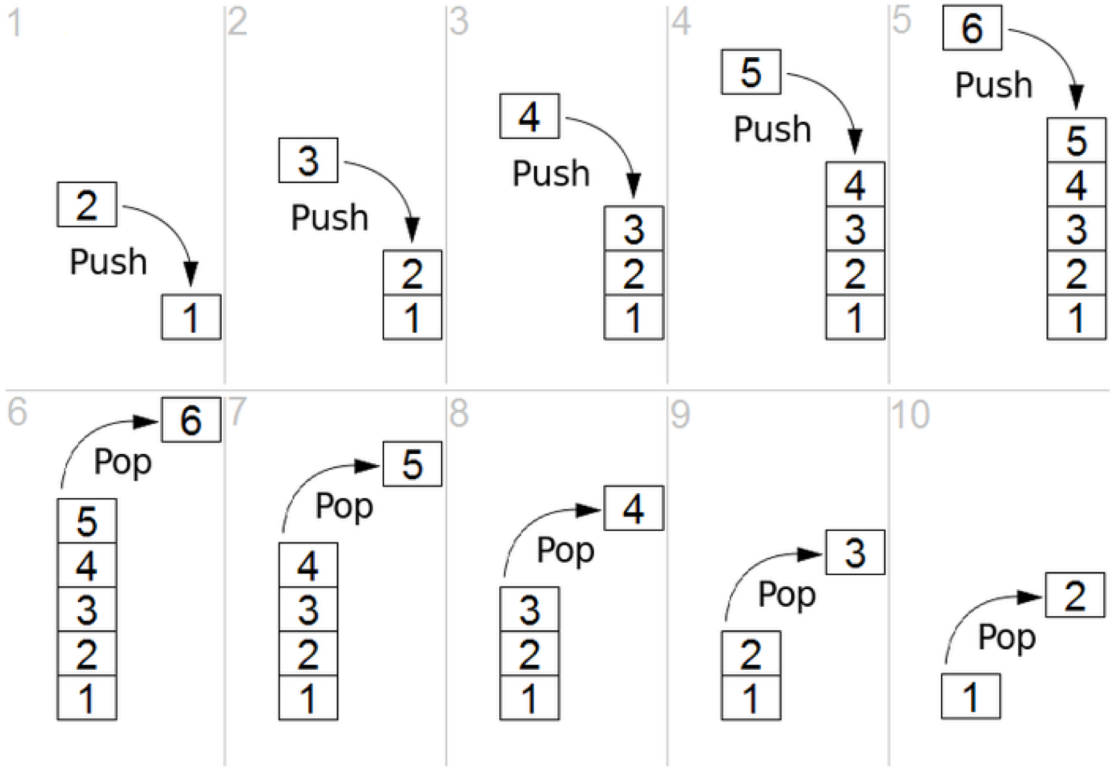
STL example

- Stack
  - serves a collection of elements with following operations:
    - **push**: which adds an element to the collection
    - **pop**: which removes the most recently added element that was not yet removed

Similar to a stack of plates, adding or removing is only possible at the top.

STL example

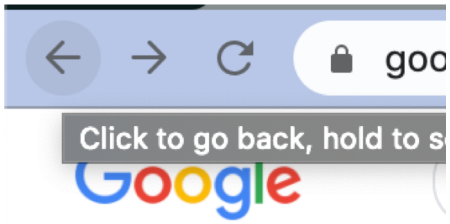
- Stack



- LIFO (Last In First Out)

STL example

- Stack
  - Usage
    - Web browser history (consider back button)
    - Undo : cancels recent job first
    - Reverse string generation
      - push every character to the stack, and pop them sequentially
  - Exceptions
    - **overflow** if exceeds maximum number of elements
    - **underflow** if tries to pop from empty stack



STL example

- Stack implementation (Vanilla) (reference: https://www.geeksforgeeks.org/stack-data-structure-introduction-program/)

```
class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
```

```
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::top()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::empty()
{
    return (top < 0);
}
```

STL example

- Implementing stack by yourself is quite inefficient
  - Use STL!!
  - methods
    - `push(element)` : push element
    - `pop()` : pop element
    - `top()` : returns the top element (without popping)
    - `size()` : returns current stack size
    - `empty()` : returns whether the stack is empty or not

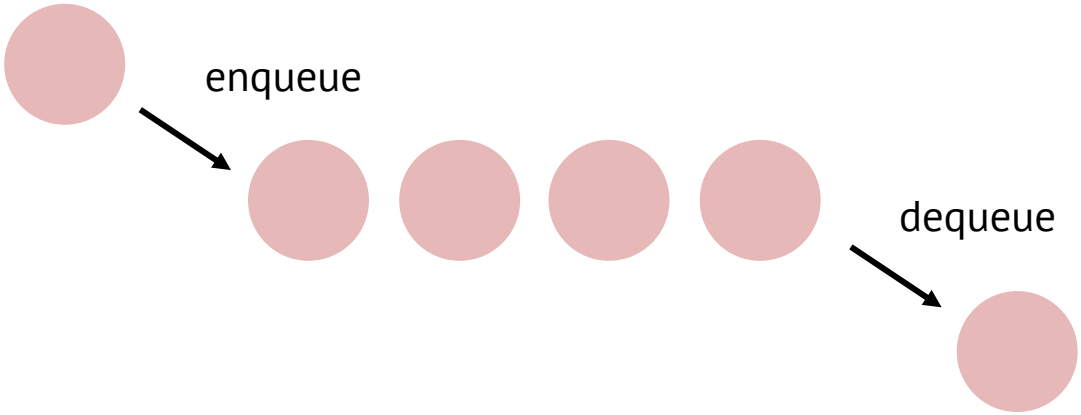
STL example

- STL stack

```
#include <stack>

stack<int> intStack;
intStack.push(1);
intStack.push(2);
intStack.push(3);

cout << intStack.top() << endl;          // 3
cout << intStack.size() << endl;         // 3

intStack.pop();

cout << intStack.size() << endl;         // 2
cout << intStack.empty() << endl;        // 0 (false)

intStack.pop();
intStack.pop();

cout << intStack.empty() << endl;        // 1 (false)
```

STL example

- Queue
  - Also a collection of elements…
  - but this time FIFO (First In First Out)
  - serves two main operations
    - enqueue: push the element at the back of queue
    - dequeue: delete the element from the front of queue



STL example

- Queue

```
class queue
{
            int *arr;
            int capacity;
            int front;
            int rear;
            int count;
public:
            queue(int size = SIZE);
            ~queue();
            void dequeue();
            void enqueue(int x);
            int size();
            bool isEmpty();
};

queue::queue(int size)
{
            arr = new int[size];
            capacity = size;
            front = 0;
            rear = -1;
            count = 0;
}

queue::~queue()
{
            delete arr;
}
```

```
void queue::dequeue()
{
            // check for queue underflow
            if (isEmpty())
            {
                        cout << "UnderFlow\nProgram Terminated\n";
                        exit(EXIT_FAILURE);
            }

            front = (front + 1) % capacity;
            count--;
}

void queue::enqueue(int item)
{
            // check for queue overflow
            if (isFull())
            {
                        cout << "OverFlow\nProgram Terminated\n";
                        exit(EXIT_FAILURE);
            }

            rear = (rear + 1) % capacity;
            arr[rear] = item;
            count++;
}


int queue::size()
{
            return count;
}

bool queue::isEmpty()
{
            return (size() == 0);
}
```

STL example

- Queue

```
class queue
{
            int *arr;
            int capacity;
            int front;
            int rear;
            int count;
public:
            queue(int size = SIZE);
            ~queue();
            void dequeue();
            void enqueue(int x);
            int size();
            bool isEmpty();
};

queue::queue(int size)
{
            arr = new int[size];
            capacity = size;
            front = 0;
            rear = -1;
            count = 0;
}

queue::~queue()
{
            delete arr;
}
```

```
void queue::dequeue()
{
            // check for queue underflow
            if (isEmpty())
            {
                        cout << "UnderFlow\nProgram Terminated\n";
                        exit(EXIT_FAILURE);
            }

            front = (front + 1) % capacity;
            count--;
}

void queue::enqueue(int item)
{
            // check for queue overflow
            if (isFull())
            {
                        cout << "OverFlow\nProgram Terminated\n";
                        exit(EXIT_FAILURE);
            }

            rear = (rear + 1) % capacity;
            arr[rear] = item;
            count++;
}


int queue::size()
{
            return count;
}

bool queue::isEmpty()
{
            return (size() == 0);
}
```

STL example

- queue methods
  - `push(element)` : enqueue (at the back)
  - `pop()` : dequeue (from the front)
  - `front()` : returns the front element
  - `back()` : returns the back element
  - `empty()` : returns whether the queue is empty or not
  - `size()` : returns current queue size

STL example

- STL queue

```
#include <queue>

queue<string> sQueue;
sQueue.push("cpp");
sQueue.push("java");
sQueue.push("javascript");
sQueue.push("python");

cout << sQueue.size() << endl;          // 4
cout << sQueue.front() << endl;         // cpp
cout << sQueue.back() << endl;          // python

sQueue.pop();
cout << sQueue.front() << endl;         // java

sQueue.pop();
cout << sQueue.size() << endl;          // 2
cout << sQueue.front() << endl;         // javascript
```

STL

- We learned three *containers*
  - vector, stack, and queue
- and corresponding methods
- However, to properly use STL containers…
- You should learn how to use
  - iterator
  - algorithms

STL iterator

- Role
  - provides general method to iterate / access elements in container
  - interface to bind container & algorithm
  - By Iterator…
    - any container & algorithm can bind itself to
    - any algorithm & container
    - (no dependency)

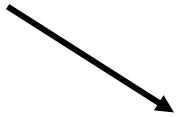
STL iterator

- vector iterator

```
vector<int> v;

v.push_back(1);
v.push_back(9);
v.push_back(5);
v.push_back(2);
v.push_back(7);

vector<int> :: iterator vIter = v.begin();

cout << vIter[2] << endl;  // random access
cout << endl;

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

output:
5

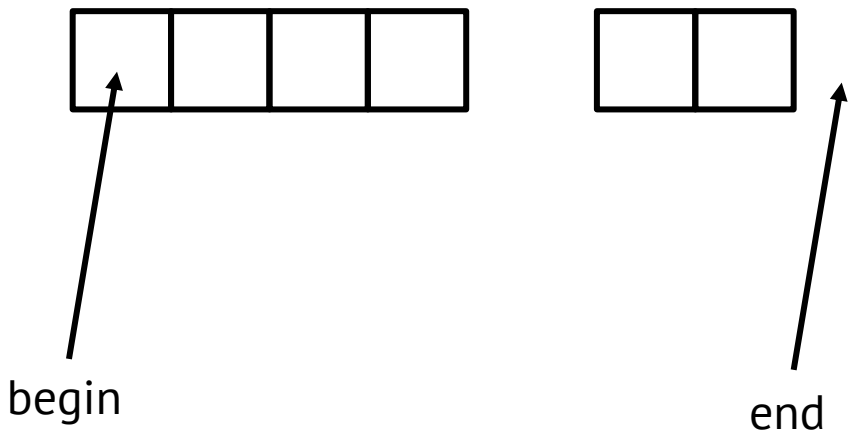
1
9
5
2
7



iterator is actually a pointer

STL Iterator

- begin, end
  - begin: directs first element
  - end: directs the location after last element



- Able to move using --, ++ operator

STL Iterator

- Random access iterator
  - powerful iterator for array-based container
    - ex) vector, deque
    - deque is a advanced version of vector (better memory usage)
  - Can also apply operators such as
    - +
    - -
    - +=, -=
    - [idx] (random access

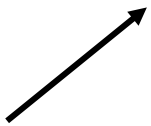
STL Algorithm

- Now we can apply powerful algorithms to STL containers
- example: vector sorting

```
#include <vector>
#include <algorithm>

v.push_back(1);
v.push_back(9);
v.push_back(5);
v.push_back(2);
v.push_back(7);

for(int i = 0; i < 5; i++) cout << v[i] << " ";   // 1 9 5 2 7
cout << endl;

sort(v.begin(), v.end());
for(int i = 0; i < 5; i++) cout << v[i] << " ";   // 1 2 5 7 9
cout << endl;

sort(v.begin(), v.end(), greater<int>());
for(int i = 0; i < 5; i++) cout << v[i] << " ";   // 9 7 5 2 1
cout << endl;
```

STL Algorithm

- Sort
  - The function that sorts the elements in the range [start, end)
  - Default: ascending order
  - usage
    - `sort(arr, arr + n)`
    - `sort(v.begin(), v.end())` : default (ascending order)
    - `sort(v.begin(), v.end(), greater<T>())` : descending order
    - `sort(v.begin(), v.end(), less<T>())` : ascending order
    - `sort(v.begin(), v.end(), compare)`



user-defined comparator

STL Algorithm

- User-defined Comparator

```
class Human {
public:
   string name;
   int age;
   Human(string name, int age) {
      this->name = name;
      this->age = age;
   }
};

bool compare(const Human& a, const Human& b) {
   return (a.age > b.age);
}


vector<Human> people;
people.push_back(Human("Alice", 12));
people.push_back(Human("Bob", 42));
people.push_back(Human("Carol", 18));
people.push_back(Human("Dave", 25));
people.push_back(Human("Eve", 40));

sort(people.begin(), people.end(), compare);
for(int i = 0; i < people.size(); i++) {
   cout << people[i].name << ", " << people[i].age << endl;
}
```

```
output:
Bob, 42
Eve, 40
Dave, 25
Carol, 18
Alice, 12
```

STL

- Much powerful if you learn
  - Template
  - Operator Overloading
- But now, just use and become familiar to it
- There exists numerous STL containers, algorithms
  - which will support your implementation
  - feel free to use them!!

STL

- However…
  - Just using it is not a good strategy
  - study underlying implementation & logic of STL
  - For example…
    - STL sort algorithm uses *quick sort*
    - *Wikipedia (*https://en.wikipedia.org/wiki/Quicksort) will help you
    - There are lots of references about STL -> use them!!
      - geeksforgeeks, stack overflow, etc…

Thank you!!

contact: jeonhyun97@postech.ac.kr