C++

程式語言（二）
Introduction to Programming (II)

Introduction to STL

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

Platform/IDE

- Dev-C++

https://www.pngegg.com/en/search?q=Dev-C

- Codeblocks

https://icons8.com/icons/set/code-blocks

- OnlineGDB (https://www.onlinegdb.com/)

- Real-Time Collaborative Online IDE (https://ide.usaco.guide/)

Textbooks (We focusing on C++11)

- ***Learn C++ Programming by Refactoring* (由重構學習 C++ 程式設計). Pang-Feng Liu (劉邦鋒). NTU Press. 2023.**
- ***C++ Primer. 5th Edition.* Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. 2019.**
- *Effective C++*. Scott Meyers. O'Reilly. 2016.
- *Thinking in C++. Vol. 1: Introducing to Standard C++*. 2nd Edition. Bruce Eckel. Prentice Hall PTR. 2000.

Useful Resources

- Tutorialspoint
  - https://www.tutorialspoint.com/cplusplus/index.htm
  - Online C++ Compiler
- Programiz
  - https://www.programiz.com/cpp-programming
- LEARN C++
  - https://www.learncpp.com/
- MIT OpenCourseWare - Introduction to C++
  - https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/pages/lecture-notes/
- Learning C++ Programming
  - https://www.programiz.com/cpp-programming
- GeeksforGeeks
  - https://www.geeksforgeeks.org/c-plus-plus/

Standard Template Library (STL)

- Standard Template Library (STL) is a powerful library in C++ that provides a collection of **generic data structures** and **algorithms** to simplify common programming tasks.
  - It is based on **templates.**
  - **Goal:** Facilitate development of reusable and efficient codes.
- Key components:
  - **Containers** (+Adapters): Data structures that store objects.
  - **Algorithms**: Predefined functions or operations.
  - **Iterators**: Objects that help traverse elements in the containers.
  - **Functor**: Classes that overload `operator()` to work as functions.

*Functor

- **Stateful**: functors can store data; it has a “state”.

```
struct Adder {
    int offset; // can keep data here
    Adder(int off) : offset(off) {}
    int operator()(int x) const { return x + offset; }
};

int main() {
    Adder add5(5);            // create a functor that adds 5
    cout << add5(10)<< endl; // calls add5.operator()(10); 15
    Adder addMinus2(-2);
    cout << addMinus2(7) << endl;  // prints 5
    return 0;
}
```

*Functor

- **Stateful**: functors can store data; it has a “state”.
- **Inlinable**: The compiler can optimize calls better through pointers-to-functions.

```
int add(int a, int b) { return a + b; }
int (*fp)(int,int); // fp is a pointer to a function
fp = &add; // or fp = add;
int result = fp(2,3); // same as add(2,3)

int apply(int x, int y, int (*op)(int,int)) { // callback?
    return op(x, y);
}
int mul(int a, int b) { return a*b; }
int main() {
    cout << apply(6, 7, add) << "\n";  // prints 13
    cout << apply(6, 7, mul) << "\n";  // prints 42
}
```

Another example:
as a comparator

*Functor

- **Stateful**: functors can store data; it has a “state”.
- **Inlinable**: The compiler can optimize calls better through pointers-to-functions.
- **Extensible:** Can be “templated” for generic behavior.

```
template<typename T>
struct Mul {
    T factor;
    Mul(T f) : factor(f) {}

    T operator()(T x) const {
        return x * factor;
    }
};
```

```
int main() {
    Mul<double> times2(2.0);
    cout << times2(3.5); // prints 7.0
    return 0;
}
```

Containers

- Sequence Containers:
  - `vector`, `deque`, `list`, `array`
- Associative Containers:
  - `set`, `map`, `multiset`, `multimap`
- Unordered Containers:
  - `unordered_set`, `unordered_map`
- Container Adapters:
  - `stack`, `queue`, `priority_queue`

Containers

- Sequence Containers:
  - `vector`, `deque`, `list`, `array`
- Associative Containers:
  - `set`, `map`, `multiset`, `multimap`
- Unordered Containers:
  - `unordered_set`, `unordered_map`
- Container Adapters:
  - `stack`, `queue`, `priority_queue`

Due to the matter of time, we will only briefly discuss some of them.

Refer to the official documents for more details.

Sequence Containers:

Elements are stored in an ordered fashion!

vector (#include<vector>)

Task	Example: `std::vector<int>`

Create a vector;
8 copies of zero;
5 copies of 3
	`vector<int> v;`
	`vector<int> v(8);`
	`vector<int> v(5, 3);`

Add `k` to the end of a vector `v`	`v.push_back(k)`
Remove the last element in a vector `v`	`v.pop_back();`
Clear the vector	`v.clear();`
Get the element at index `i`	`w = v.at(i);` `w = v[i];`

Note:

```
std::vector<int> v = { 9, 7 };
v[2] = 5; // undefined; out of bound
```

vector (contd.)

Task	Example: `std::vector<int>`
Check if the vector is empty	`if (v.empty())`
Insert `w` at some index `i` of the vector	`v.insert(v.begin()+i, k)`
Remove the element at index `i` of the vector	`v.erase(v.begin()+i)`
Get the sublist in indices `[i, j)`	`vector<int>c(v.begin()+i, v.begin()+j);`
Request capacity for a vector	`v.reserve(100000);`

Note:
`v.begin()` and `v.end()` are iterators (we will introduce them later).

Benefit of `vector<T>.reserve()`

- **Example**: Create a vector of a large number of integers.

```
std::vector<int> v;
for (size_t i = 0; i < 1000000; ++i) {
    v.push_back(i);
}
```

⇕

```
std::vector<int> v;
v.reserve(1000000);
for (size_t i = 0; i < 1000000; ++i) {
    v.push_back(i);
}
```

deque: Similar to vector (#include<deque>)

- `deque` (double-ended queue) supports faster insertion anywhere.

```
deque<int> dq{3, 4}; // {3, 4}
dq.push_front(2); // {2, 3, 4}
dq.pop_back(); // {2, 3}
dq[1] = 0; // {2, 0}
```

Note:
`deque` has `push_front()` and `pop_front()`, while `vector` doesn't.

Container Adapters

- A wrapper of an (other STL) object that changes **how external users can interact with that object.**
- It modifies or restricts the **interface** of existing sequence containers (`vector`, `deque`, `list`) to provide a **specialized behavior**.
- Examples: `stack, qeque, priority_queue`
  - Will be introduced in *Data Structures* course in detail.

Stack (#include<stack>)

https://cplusplus.com/reference/stack/stack/

- Implementation of LIFO (last-in-first-out) structure.

```
std::stack<T>
```

class template

std::stack <stack>

```
template <class T, class Container = deque<T>> class stack;
```

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

stacks are implemented as ***container adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed***/***popped*** from the "***back***" of the specific container, which is known as the ***top*** of the stack.

Queue (#include<queue>)

https://cplusplus.com/reference/queue/queue/

- Implementation of FIFO (first-in-first-out) structure.

std::queue<T>

class template

std::queue <queue>

```
template <class T, class Container = deque<T> > class queue;
```

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

priority_queue
(#include<queue>)

- Implementation of heap-based priority queue structure.
  - Elements are ranked based on a certain priority.

    `std::priority_queue<T>`
  - It supports constant time lookup at the expense of logarithmic time of insertion and deletion.

priority_queue

https://cplusplus.com/reference/queue/priority_queue/

class template

std::priority_queue

<queue>

```
template <class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;
```

Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some ***strict weak ordering*** criterion.

This context is similar to a ***heap***, where elements can be inserted at any moment, and only the ***max heap*** element can be retrieved (the one at the top in the ***priority queue***).

Priority queues are implemented as ***container adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***popped*** from the ***"back"*** of the specific container, which is known as the ***top*** of the priority queue.

Illustrating Examples

class template

std::stack

```
template <class T, class Container = deque<T> > class stack;
```

```
// Container = std::deque
std::stack<int> stack_deq;
// Container = std::vector
std::stack<int, std::vector<int>> stack_vec;
// Container = std::list
std::stack<int, std::list<int>> stack_list;
```

Associative Containers:

Elements are organized and managed automatically using **keys**!

Elements are stored using a specific ordering mechanism so that it’s more efficient for searching, insertion and deletion

map (#include<map>)

Task	Example: `std::map<int, char>`
Create a map	`map<int, char> m;`
Add key `k` with value `v` into the map	`m.insert({k, v});` `m[k] = v;`
Remove key `k` from the map	`m.erase(k);`
Check if key `k` is in the map	`if (m.count(k))`
Check if the map is empty	`if (m.empty())`
Retrieve or overwrite value associated with key `k`	`char c = m[k];` `m[k] = v;`

Note: Actually, the underlying type stored in `std::map<K, V>` is
`std::pair<const K, V>`

Comparison operator is required for map (also for set)

```
class Person {
public:
    std::string name;
    int age;
    Person(std::string n, int a) : name(n), age(a) {}
};
```

```
std::map<int, int> map1; // OKAY - comparable
std::map<Person, int> map2; // ERROR - not comparable
// No operator < defined for Person!
```

A Solution

```
struct Person {
    std::string name;
    int age;
    Person(std::string n, int a): name(n), age(a) {}

    bool operator<(Person const& other) const {
        if (name < other.name)        // compare names first
            return true;
        else if (other.name < name)   // if this name is greater
            return false;
        else                          // names are equal, compare ages
            return age < other.age;
    }
};
```

https://onlinegdb.com/nlMMCpknXM

Iterators

Have a look at looping over a collection

```
std::vector<int> v{5, 4, 3, 2, 1, 0};
for (size_t i=0; i < v.size(); i++) { // looks good!
    const auto& e = v[i];
    cout << e << endl;
}
// The following way looks great, too.
for (auto &e: v) {
    cout << e << endl;
}
// But how about looping over a set or a map?
// some_element++?? What does ++ mean here?
```

Note: Structured binding is available in C++17, not C++11.

```
std::map<int, char> m;
for (const auto& [key, value]: m) // structure binding here
    // work with key, value ...
```

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Iterators

```
auto it = v.begin()
```

- Iterator can help in iterating over ANY container.
- It becomes more flexible and consistent to traverse and manipulate elements in a container (say `v`).
- The iterator always knows “the next element”.

Why use `++it` instead of `it++`?

- `++it` increments the iterator first and then returns the updated iterator.
- `it++` returns the current iterator first, and then increments it.
  - An extra step is required here! We have to store a copy of the original iterator!

- However, it still depends on the scenario.

Example: Printing all elements in a map

```
std::map<int, int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
    const auto& [key, value] = *iter; // structured binding!
    cout << key << ":" << value << endl;
}
```

⇩

```
std::map<int, int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
    const auto& key = (*iter).first; //resolving the issue in C++11
    const auto& value = (*iter).second;
    cout << key << ":" << value << endl;
}
```

Example: 3Sum (LeetCode)

https://leetcode.com/problems/3sum/

15. 3Sum

Solved

Medium | Topics | Companies | Hint

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
```

Example 2:

```
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
```

Example 3:

```
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.
```

Code

C++ | Auto

```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {

    }
};
```

Example: 3Sum (LeetCode)

- A brute-force solution

```
class Solution {
public:
     vector<vector<int>> threeSum(vector<int>& nums) {
          int len = nums.size();
          if(len==0 || len<3) // at least three elements needed
               return {};
          set<vector<int>> s; // "set" stores unique elements only
                              // duplicate triplets will not be added
          sort(nums.begin(),nums.end());
          // Compare all cases of group of three elements
          for(int i=0;i<len-2;i++){
               for(int j=i+1;j<len-1;j++){
                    for(int k=j+1;k<len;k++){
                         if(nums[i]+nums[j]+nums[k]==0){
                              s.insert({nums[i],nums[j],nums[k]});
                         }
                    }
               }
          }
          // Insert all unique triplets in result vector
          vector<vector<int>> ans(s.begin(),s.end());
          return ans;
     }
};
```

Example: 3Sum (LeetCode)

https://leetcode.com/problems/3sum/

- https://onlinegdb.com/q5n8kg7ku

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

using namespace std;
```

- `upper_bound()`:
  - find the **first** element in a sorted range that is **strictly greater** than a given value

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

using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        //preprocessing
        int countPos = 0, countNeg = 0, countZero = 0;
        for (auto it = nums.begin(); it != nums.end(); ++it) {
            if (*it > 0) {
                countPos = 1;
            } else if (*it < 0) {
                countNeg = 1;
            } else { //  *it == 0
                countZero++;
            }
        }
        // after preprocessing
        vector<int>::iterator start, fpos, searchEnd;
        vector<vector<int>> result = {};
        if (countPos + countNeg < 2) {
            if (countZero > 2) return {{0,0,0}};
            else return {};
        }
        sort(nums.begin(), nums.end());
        for (auto start = nums.begin(); start < nums.end()-2; ++start) {
            searchEnd = nums.end();
            if (start > nums.begin() && *start == *(start-1)) continue;
            for (auto mid = start+1; mid < searchEnd - 1; ++mid) {
                if (mid > start+1 && *mid == *(mid-1)) continue;
                fpos = upper_bound(mid+1, searchEnd, -(*start)-(*mid));
                if (*(fpos-1) == -(*start)-(*mid) && fpos>mid+1) {
                    result.push_back({*start, *mid, -(*start)-(*mid)});
                    searchEnd = fpos;
                } //else searchEnd -= 1;
                if (fpos<=mid+1) break;
            }
        }
        return result;
    }
};
```

Example: 3Sum (LeetCode)

https://leetcode.com/problems/3sum/

- https://onlinegdb.com/q5n8kg7ku

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

using namespace std;
```

- `upper_bound()`:
  - find the **first** element in a sorted range that is **strictly greater** than a given value

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

using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        //preprocessing
        int countPos = 0, countNeg = 0, countZero = 0;
        for (auto it = nums.begin(); it != nums.end(); ++it) {
            if (*it > 0) {
                countPos = 1;
            } else if (*it < 0) {
                countNeg = 1;
            } else { //  *it == 0
                countZero++;
            }
        }
```

Example: 3Sum (LeetCode)

https://leetcode.com/problems/3sum/

- https://onlinegdb.com/q5n8kg7ku

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

using namespace std;
```

- `upper_bound()`:
  - find the **first** element in a sorted range that is **strictly greater** than a given value

```
        // after preprocessing
        vector<int>::iterator start, fpos, searchEnd;
        vector<vector<int>> result = {};
        if (countPos + countNeg < 2) {
            if (countZero > 2) return {{0,0,0}};
            else return {};
        }
        sort(nums.begin(), nums.end());
        for (auto start = nums.begin(); start < nums.end()-2; ++start) {
            searchEnd = nums.end();
            if (start > nums.begin() && *start == *(start-1)) continue;
            for (auto mid = start+1; mid < searchEnd - 1; ++mid) {
                if (mid > start+1 && *mid == *(mid-1)) continue;
                fpos = upper_bound(mid+1, searchEnd, -(*start)-(*mid));
                if (*(fpos-1) == -(*start)-(*mid) && fpos>mid+1) {
                    result.push_back({*start, *mid, -(*start)-(*mid)});
                    searchEnd = fpos;
                } //else searchEnd -= 1;
                if (fpos<=mid+1) break;
            }
        }
        return result;
    }
};
```

Example: 3Sum (LeetCode)

https://leetcode.com/problems/3sum/

- https://onlinegdb.com/q5n8kg7ku

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

using namespace std;
```

- `upper_bound()`:
  - find the **first** element in a sorted range that is **strictly greater** than a given value

Example: 3Sum (LeetCode)

- Can be even efficient.
- Try on your own.

Discussions & Questions