Advanced C++ STL

Tony Wong
2017-03-25

C++ Standard Template Library

- Algorithms
  - Sorting
  - Searching...
- Data structures
  - Dynamically-sized array
  - Queues...
- The implementation is abstracted away from the users
  - The implementation may differ from compilers, and may change over time (to improve performances)

What is a template?

- Class / function template can be applied to different data types

```
template<class T> T triple(T x) {
  return x + x + x;
}
```

- The type T will be determined at *compile time*, and for each different type, a function for that type will be created

```
triple(1)   triple(2.3)   triple(true)   triple(string{"abc"})
T = int     T = double    T = bool       T = string
result: 3   result: 6.9   result: true   result: abcabcabc
```

```
int triple(int x) {
  return x + x + x;
}
```

What is a template?

- Another function template example

```
template<class T> T absmax(T a, T b) {
  return abs(a) > abs(b) ? a : b;
}
```

Can compile:

```
absmax(-1, -2);
absmax('a', 'b');
```

Compilation error:

```
absmax(1.5, 3);
absmax(string{"a"}, string{"b"});
```

What is a template?

- Class template example:

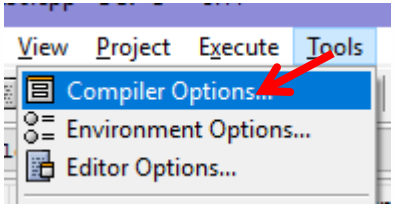
```
template<class T>
struct MCChoices {
  T a, b, c, d;
};
template<class T>
ostream& operator<<(ostream& os, const MCChoices<T>& t) {
  os << "A. " << t.a << endl;
  os << "B. " << t.b << endl;
  os << "C. " << t.c << endl;
  os << "D. " << t.d << endl;
  return os;
}
```

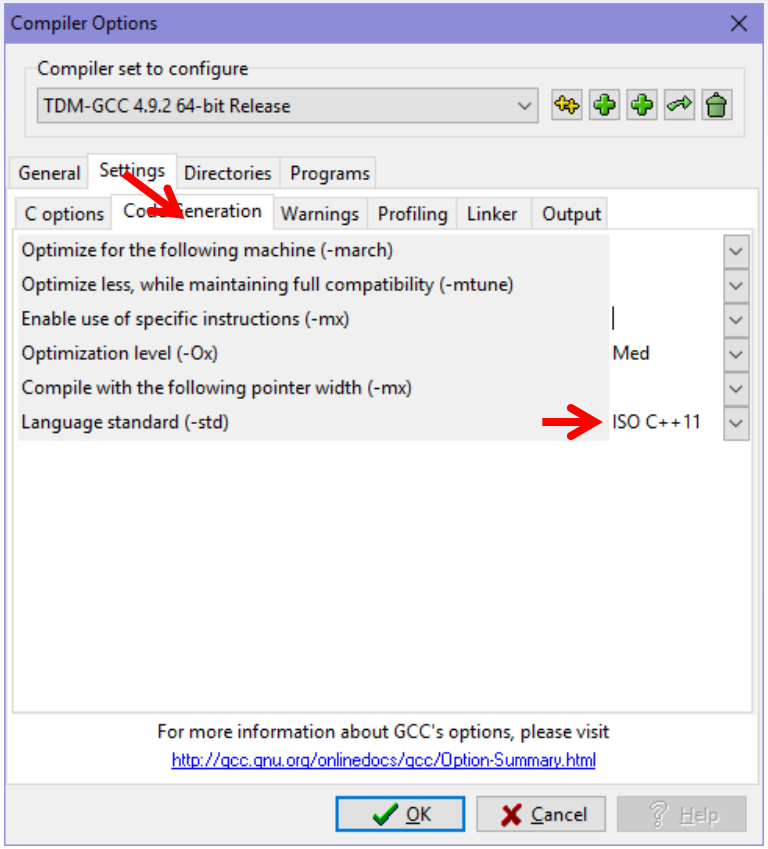
```
cout << MCChoices<int>{2, 4, 8, 16};
cout << MCChoices<string>{
  "i, ii only",
  "i, iii only",
  "ii, iii only",
  "i, ii and iii"
};
```

```
E:\OneDrive\test60c.exe
A. 2
B. 4
C. 8
D. 16
A. i, ii only
B. i, iii only
C. ii, iii only
D. i, ii and iii
```

Enabling C++11

- Some features introduced here are only available in C++11





<algorithm> sort

Best case: $O(n)$
Avg/Worst case: $O(n \lg n)$

where $n = r - l$

- **`sort(RndAccIt l, RndAccIt r);`**
- sorts the range **`[l, r)`** in non-decreasing order
- where **`l`** and **`r`** are **Random Access Iterators**
  - Pointers are also random access iterators

```
1  #include <iostream>
2  #include <algorithm>
3  int a[10] = {4, 7, 1, 8, 2, 5, 4, 9, 6, 3};
4  int main() {
5      std::sort(a, a + 10);
6      for (int i = 0; i < 10; i++) {
7          std::cout << a[i] << ' ';
8      }
9      return 0;
10 }
```

Result: 1 2 3 4 4 5 6 7 8 9

<algorithm> sort

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int a[10];
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a, a + n);
    for (int i = 0; i < n; i++) {
       cout << a[i] << ' ';
    }
    return 0;
}
```

Cause all identifiers defined in the **`std`** namespace usable without using **`std::`**

```cpp
int a[11];
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    sort(a + 1, a + n + 1);
    for (int i = 1; i <= n; i++) {
       cout << a[i] << ' ';
    }
```

Input: 1 4 2 8 5 7
Output: 1 2 4 5 7 8

<algorithm> sort

- To sort user-defined types, you need to "teach" **`sort`** how to compare objects:
  - Implement **`operator<`**
    `bool operator<(const T& o) const`

Method 1:

```
4  struct Student {
5    string name;
6    double height;
7    bool operator<(const Student& o) const {
8      return height > o.height;
9    }
10 };
```

Can be omitted if you are lazy

Sorts in descending order instead

Method 2:

```
struct Student {
  string name;
  double height;
};
bool operator<(const Student& a, const Student& b) {
  return a.height > b.height;
};
```

<algorithm> sort

- Alternatively, you can implement function and instructs **sort** to use it
  bool cmp(const T& a, const T& b)

you can use any name

```
struct Student {
  string name;
  double height;
};
bool cmp(const Student& a, const Student& b) {
  return a.height > b.height;
};
......
  sort(students, students + n, cmp);
```

!!! Be careful !!!
The comparison function should return **true** *if and only if* object **a** ***must*** be placed before **b**.
If your function returns **true** for both **cmp(x, y)** and **cmp(y, x)** you will get runtime error TLE because the algorithm tries to place **x** before **y** and place **y** before **x** (contradiction)
This also applies to the variant in the previous slide

<algorithm> sort

- With C++11 you can also write an anonymous function
- The return type **bool** will be inferred by the compiler

```
sort(students, students + n,
  [](const Student& a, const Student& b) {
    return a.height > b.height;
  });
```

<algorithm> stable_sort

Worst case: $O(n \lg n)$

- `stable_sort(RndAccIt l, RndAccIt r)`
- It only has effect when some data is not part of the sorting key, e.g. `string name`
- For details, read Sorting and Searching

```
#define N 100000000
int a[N];
int main() {
  for (int i = 0; i < N; i++) {
    a[i] = N - i;
  }
  sort(a, a + N);
```

1.38s

```
#define N 100000000
int a[N];
int main() {
  for (int i = 0; i < N; i++) {
    a[i] = N - i;
  }
  stable_sort(a, a + N);
```

3.08s

<algorithm> nth_element

Average case: $O(n)$
Worst case: $O(n \lg n)$

- `nth_element(RndAccIt l, RndAccIt nth, RndAccIt r)`
- Implements the quick select algorithm: $O(n)$
- Reorders the range `[l, r)` such that
  - The element at `nth` will contain the same value if `[l, r)` is sorted
  - All elements in `[l, nth)` is less than or equal to that in `nth`
  - The element `nth` less than or equal to those in `(nth, r)`

```
int a[10] = {4, 7, 1, 8, 2, 5, 4, 9, 6, 3};
int main() {
    std::nth_element(a, a + 8, a + 10);
    for (int i = 0; i < 10; i++) {
        cout << a[i] << ' ';
    }
```

Possible output:
5 3 1 4 2 4 7 6 **8** 9

<algorithm> reverse

$O(n)$

- **`reverse(BiDirIt l, BiDirIt r)`**
- Obviously, reverses the range **`[l, r)`**

```
int a[10] = {4, 7, 1, 8, 2, 5, 4, 9, 6, 3};
sort(a, a + n);
reverse(a, a + n);
```

- Now array a becomes `{9, 8, 7, 6, 5, 4, 4, 3, 2, 1}`

```
char s[] = "Hello, World!";
reverse(s, s + strlen(s));
cout << s << endl;
```

- Result: **`!dlroW ,olleH`**

<algorithm> unique

$O(n)$

- **`FwdIt unique(FwdIt l, FwdIt r)`**
- Removes consecutive equal elements and returns an ForwardIterator / pointer to the end of the range
- Therefore we can get the new length by substracting the returned pointer by the beginning of the range

```
char s[] = "abcccdddcccbbaa";
int newlen = unique(s, s + strlen(s)) - s;
s[newlen] = 0;
// *unique(s, s + strlen(s)) = 0
cout << s << endl;
```

Result: **abcdcba**

- To get the distinct elements in a range, **`sort`** the range first then apply **`unique`**

<algorithm> binary_search

RndAccIt: $O(\lg n)$
$O(n)$ otherwise

- **`bool binary_search(FwdIt l, FwdIt r, val)`**
- Searches the range **`[l, r)`** for the value **`val`**
- The range **`[l, r)`** must be either:
  - Sorted in non-descending order
  - *Partitioned* with respect to **`val`**


```
int a[10] = {1, 3, 5, 5, 5, 7, 9, 11, 11, 13};
cout << binary_search(a, a + 10, 5) << endl; 1
cout << binary_search(a, a + 10, 8) << endl; 0
int b[10] = {5, 4, 3, 2, 1, 6, 10, 9, 8, 7};
cout << binary_search(b, b + 10, 6) << endl; 1
cout << binary_search(b, b + 10, 3) << endl; ?
```

Incorrect usage

<algorithm> lower_bound

RndAccIt: $O(\lg n)$
$O(n)$ otherwise

- **`FwdIt lower_bound(FwdIt l, FwdIt r, val)`**
- Finds the left-most position that **`val`** can be inserted into the range such that the range remains sorted/partitioned
- Equivalently, finds the left-most element that is **`>= val`**

```
int a[10] = {1, 3, 5, 5, 5, 7, 9, 11, 11, 13};
cout << distance(a, lower_bound(a, a + 10, 5)) << endl;   2
cout << distance(a, lower_bound(a, a + 10, 8)) << endl;   6
int b[10] = {5, 4, 3, 2, 1, 6, 10, 9, 8, 7};
cout << lower_bound(b, b + 10, 20) - b << endl;           10
```

If every element in the range is **`< val`**, **`r`** is returned

<algorithm> upper_bound

RndAccIt: $O(\lg n)$
$O(n)$ otherwise

- **`FwdIt upper_bound(FwdIt l, FwdIt r, val)`**
- Finds the right-most position that **`val`** can be inserted into the range such that the range remains sorted/partitioned
- Equivalently, finds the left-most element that is **`> val`**

```
int a[10] = {1, 3, 5, 5, 5, 7, 9, 11, 11, 13};
cout << distance(a, upper_bound(a, a + 10, 5)) << endl;    5
cout << distance(a, upper_bound(a, a + 10, 8)) << endl;    6
```

Note: **`distance`** is $O(1)$ for RndAccIt, $O(n)$ for FwdIt

<algorithm> next_permutation

$O(n)$
$O(1)$ amortized

- **`bool next_permutation(1, r)`**
- Rearranges the range to form the next *lexicographically greater* permutation
- If the range already contains the greatest permutation (elements in non-increasing order), **`false`** is returned and the range is rearranged to form the smallest permutation (elements in non-decreasing order)

```
int a[4] = {3, 4, 2, 2};
bool ret = next_permutation(a, a + 4);
for (int i = 0; i < 4; i++) {
  cout << a[i] << ' ';
}
cout << ": " << ret << endl;
int b[4] = {4, 3, 2, 2};
ret = next_permutation(b, b + 4);
for (int i = 0; i < 4; i++) {
  cout << b[i] << ' ';
}
cout << ": " << ret << endl;
```

Result:
4 2 2 3 : 0

Result:
2 2 3 4 : 1

```
int c[5] = {1, 2, 3, 4, 5};
do {
  for (int i = 0; i < 5; i++) {
    cout << c[i] << (i == 4 ? '\n' : ' ');
  }
} while (next_permutation(c, c + 5));
```

Result:
1 2 3 4 5
1 2 3 5 4
1 2 4 3 5
....
5 4 3 1 2
5 4 3 2 1

string

- When writing modern C++ programs, avoid using arrays, C strings and raw pointers

```
string s;
cout << "s is \"" << s << "\"" << endl;
cin >> s;
cout << "s is \"" << s << "\"" << endl;
```

creates an empty string

stops at space, new line or tab

Input:
```
abc def
```

Output:
```
s is ""
s is "abc"
```

- string uses 0-based indexing

```
string t = "ghi";
string u{"jkl"};
cout << t << u << endl;
t[1] = 'x';
cout << t << endl;
```

{} or () both ok

Output:
```
ghijkl
gxi
```

string

Append a character

```
string s = "abcd";
s += 'e';
cout << s << endl;
```

Output: abcde

Concatenate

```
string s;
string t = "abcdefghijklmnopqrstuvwxy";
for (int i = 0; i < 20000; i++) {
  s = s + t;
}
cout << s.length() << endl;
```

Output: 500000
Time: 0.67 sec

Append

```
string s;
string t = "abcdefghijklmnopqrstuvwxy";
for (int i = 0; i < 20000; i++) {
  s += t;
}
cout << s.length() << endl;
```

Output: 500000
Time: 0.02 sec

string

- Reading a whole line including spaces:

```
string s;
getline(cin, s);
```

Input:
abc def
Output:
abc def

- Get substring from
  - `s.substr(index, length)`

```
string s = "abcdefg";
s.substr(1, 4)  // "bcde"
s.substr(5, 10) // "fg"
```

- Compare strings (lexicographically)
  - Comparison operators: `<` `<=` `==` `>=` `>` `!=`
  - `s.compare(t)` returns negative number when `s < t`, 0 when `s == t`, positive number when `s > t`

```
string{"abc"}.compare("def")
string{"123"}.compare("123")
string{"xyz"}.compare("XYZ")
```

- Convert C++ string into null-terminated C string: `s.c_str()`

string

- **`s.begin()`** returns a RndAccIt of type **`string::iterator`** that points to the first character
- **`s.end()`** returns a RndAccIt that points to position past the last character
- Reverse a string

```
string s = "abcdef";
reverse(s.begin(), s.end());
cout << s << endl; // fedcba
```

- Reverse a substring

```
string s = "abcdef";
reverse(s.begin() + 2, s.begin() + 5);
cout << s << endl; // abedcf
```

- Sort the characters in a string

```
string s = "google";
sort(s.begin(), s.end());
cout << s << endl; // eggloo
```

Exercise - G091BB

- Input: a number N
- Output: output the next number greater than N that contains the same number of 1s, 2s, ... 9s
  - Note: you can change the number of 0s

Input	Output
4	
115	Case #1: 151
1051	Case #2: 1105
6233	Case #3: 6323
321	Case #4: 1023

Exercise - G091BB

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int t;
  cin >> t;
  for (int i = 1; i <= t; i++) {
    string s;
    cin >> s;
    cout << "Case #" << i << ": ";
    if (next_permutation(s.begin(), s.end())) {
      cout << s << endl;
    } else {
      iter_swap(upper_bound(s.begin(), s.end(), '0'), s.begin());
      cout << s[0] << 0 << s.substr(1) << endl;
    }
  }
  return 0;
}
```

Next permutation exists

Swap the elements pointed by the iterators

Find position of first non-'0' element

Containers

- Sequence containers
  - **`vector`**, **`list`**, **`deque`**
- Associative containers
  - **`set`**, **`map`** (and **`multi-`**variant)
- Unordered associative containers
  - **`unordered_set`**, **`unordered_map`**
- Adapters
  - **`queue`**, **`stack`**, **`priority_queue`**
- Automatic memory management

vector<T>

- Dynamically sized array
- Always prefer vector to raw arrays
- To declare an empty **int** array

```
vector<int> a;
```

- To declare a zero-initialized **long long** "array" of size 100

```
vector<long long> a(100);
```

- To declare an **int** "array" of size n, with element initialized to 100

```
int n;
cin >> n;
vector<int> a(n, 100);
cout << a[n - 1] << endl; // 100
```

vector<T>

- Unlike arrays, **`vector`** is aware of its size

curly braces initializer

```
vector<int> a{1, 2, 3, 4, 5};
cout << a.size() << endl; // 5
a.push_back(10);
cout << a[5] << endl; // 10
cout << a.size() << endl; // 6
```

append an item to the end

$O(1)$ amortized

```
vector<int> a;
a[0] = 123;  // runtime error is very likely
a[-1] = 123; // runtime error is very likely
```

vector<T>

- When there is no capacity left, another memory of size 2 times the original capacity is allocated. The capacity is now doubled and the data is moved to the newly allocated memory.

- After n **push_back**s , the elements are moved at most $2n$ times

```
vector<int> a;
cout << "Initial capacity: " << a.capacity() << endl;
for (int i = 1; i <= 10; i++) {
  a.push_back(i);
  cout << a.capacity() << ' ';
}
```

Output:

```
Initial capacity: 0
1 2 4 4 8 8 8 8 16 16
```

vector<T>

Traversing using indices

```
vector<int> a{1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < int(a.size()); i++) {
  sum += a[i];
}
for (size_t i = 0; i < a.size(); i++) {
  sum += a[i];
}
for (int i = int(a.size()) - 1; i >= 0; i--) {
  sum += a[i];
}
for (size_t i = a.size(); i-- > 0;) {
  sum += a[i];
}
cout << sum << endl; // 60
```

Traversing using iterators

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
  sum += *it;
}
for (auto it = a.begin(); it != a.end(); ++it) {
  sum += *it;
}
```

C++11 range-based for loop

```
for (int x : a) {
  sum += x;
}
for (auto x : a) {
  sum += x;
}
```

vector<T>

- Buggy bubble sort

```
Message
In function 'int main()':
[Warning] comparison between signed and unsigned integer expressions [-Wsign-compare]
[Warning] comparison between signed and unsigned integer expressions [-Wsign-compare]
```

- `.size()` returns an unsigned integer

```
vector<int> a;
cout << a.size() - 1 << endl;
// 18446744073709551615 for 64-bit
// 4294967295 for 32-bit
```

- Possible fixes:

```
for (int i = 1; i < int(a.size()) - 1; i++) {
  for (int j = 0; j < int(a.size()) - i; j++) {
```

```
for (int i = 1; i + 1 < a.size(); i++) {
  for (int j = 0; j + i < a.size(); j++) {
```

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i < a.size() - 1; i++) {
    for (int j = 0; j < a.size() - i; j++) {
      if (a[j] > a[j + 1]) {
        swap(a[j], a[j + 1]);
      }
    }
  }
  for (int i = 0; i < n; i++) {
    cout << a[i] << (i == n - 1 ? '\n' : ' ');
  }
  return 0;
}
```

vector<T>

- Sorting and binary search

For vector,
.begin() and .end() return
RandomAccessIterators

```
vector<int> a{1, 4, 2, 8, 5, 7};
sort(a.begin(), a.end()); // 1 2 4 5 7 8
vector<int>::iterator it = lower_bound(a.begin(), a.end(), 2);
auto it2 = upper_bound(a.begin(), a.end(), 10);
cout << distance(a.begin(), it) << endl;  // 1
cout << distance(a.begin(), it2) << endl; // 6
cout << *it2 << endl; // some strange value / runtime error
if (it2 != a.end() && *it2 == 10) {
  cout << "Found" << endl;
} else {
  cout << "Not found" << endl;
}
```

don't do this →

vector<T>

- 2D vector

```
int n, m;
cin >> n >> m;
vector<vector<int>> a(n, vector<int>(m));
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    cin >> a[i][j];
  }
}
```

vector<T>

$O(n)$

- **vector**s can be compared directly (lexicographical order) using comparison operators **< <= == >= > !=**

```
vector<int> a{1, 2, 3};
vector<int> b{1, 2, 3};
vector<int> c{1, 2, 4};
vector<int> d{1, 2, 3, 0};
cout << (a == b) << endl; // 1
cout << (a > c) << endl;  // 0
cout << (b != c) << endl; // 1
cout << (a < d) << endl;  // 1
```

vector<T>

- Performance
  - array: 3.62 sec
  - vector: 3.73 sec

```
int n = 2000;
int a[n][n];
//vector<vector<int>> a(n, vector<int>(n));
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    a[i][j] = i * n + j;
  }
}
long long sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      sum += a[j][k];
    }
  }
}
cout << sum << endl;
// 15999996000000000
```

pair<T1, T2>

- **pair** is a type template defined in **<utility>**
- A **pair<T1, T2>** stores two data:
  - **first** of type **T1**
  - **second** of type **T2**
- For example, we can use a **pair<string, int>** to store a person's height and his/her height

```
pair<string, int> p;
p.first = "Sowon";
p.second = 173;
```

```
pair<string, int> p{"Sowon", 173};
```

pair<T1, T2>

- **pair**s can be compared directly using comparison operators **< <= == >= > !=** if both **first** can **second** can be compared (have **operator<** implemented)
- Therefore, a **vector<pair<string, int>>** can be sorted directly without writing custom comparison function

```
vector<pair<string, int>> members{
  {"Sowon", 173}, {"Yerin", 167},
  {"Eunha", 163}, {"Yuju", 169},
  {"SinB", 165}, {"Umji", 163},
};
sort(members.begin(), members.end());
for (auto& p : members) {
  cout << p.first << " " << p.second << endl;
}
```

```
Eunha 163
SinB 165
Sowon 173
Umji 163
Yerin 167
Yuju 169
```

(sorted by name)

list<T>

- Implements bidirectional linked list

```
list<int> l{10, 20};
l.push_back(30);
l.push_front(0);
cout << l.front() << endl; // 0
cout << l.back() << endl;  // 30
auto it = l.begin();    // >0  10  20  30
++it;                   //  0 >10  20  30
it = l.erase(it);       //  0 >20  30
++it;                   //  0  20 >30
it = l.insert(it, 25); //  0  20 >25  30
l.insert(l.end(), 40); //  0  20 >25  30  40
cout << *it << endl;    // 25
```

`it = l.insert(it, x)` $O(1)$

Inserts **`x`** before the element pointed by **`it`**

Returns an iterator pointing to the element inserted

`it = l.erase(it)` $O(1)$

Remove the element pointed by **`it`**

Returns an iterator pointing to the next element after **`it`**

list<T>

- Traverse the list in reverse

```
list<int> l{1, 4, 2, 8, 5, 7};
for (auto it = l.rbegin(); it != l.rend(); ++it) {
  cout << *it << ' ';
} // 7 5 8 2 4 1
```

- Sorting a linked list

$O(n \lg n)$

  - You cannot use **`sort(l.begin(), l.end())`** because **`l.begin()`** and **`l.end()`** return BiDirIt instead of RndAccIt
  - Instead, call the specialized version **`l.sort()`**

```
list<int> l{1, 4, 2, 8, 5, 7};
l.sort();
```

deque<T>

- Double-ended Queue
- $O(1)$ **`push_front / pop_front / push_back / pop_back`**
- $O(1)$ random access to any element

```
deque<int> dq{4, 5, 6};
dq.push_front(3);
dq.push_back(7);
cout << dq[2] << endl; // 5
dq.pop_front();
cout << dq[2] << endl; // 6
```

queue<T>

- An adapter of **`deque`** with:
  - **`push_back(x)`** becomes **`push(x)`**
  - **`pop_front()`** becomes **`pop()`**
  - Random access removed
  - No iterators, no range-based loop
  - No curly braces initializer
- Note: **`list`** can also be used as the underlying container

stack<T>

- An adapter of **`deque`** with:
  - **`push_back(x)`** becomes **`push(x)`**
  - **`pop_back()`** becomes **`pop()`**
  - **`back()`** becomes **`top()`**
  - Random access and **`front()`** removed
  - No iterators, no range-based loop
  - No curly braces initializer
- Note: **`list`** and **`vector`** can also be used as the underlying container

vector / list / deque comparison

	`vector`	`list`	`deque`
Random Access [ ]	O(1)	No	O(1)
push / pop front	O(n)	O(1)	O(1)
push / pop back	O(1)	O(1)	O(1)
Iterators	RndAccIt	BiDirIt	RndAccIt
Insert / erase at iterator	O(n)	O(1)	O(n)
Sort	O(n lg n)	O(n lg n)	O(n lg n)
Binary search	O(lg n)	No	O(lg n)
Contiguous memory	Yes	No	No

vector / deque?

- The data inside vector is always stored in contiguous memory

```
//vector<int> v;
deque<int> v;
for (int i = 0; i < 10000; i++) {
  v.push_back(i);
}
long long sum = 0;
for (int i = 0; i < 100000; i++) {
  for (int j = 0; j < 10000; j++) {
    sum += v[j];
  }
}
cout << sum << endl;
```

vector
0.42s

```
004717a3 <+163>: movslq (%rax),%rcx
004717a6 <+166>: add    $0x4,%rax
004717aa <+170>: add    %rcx,%rdx
004717ad <+173>: cmp    %r8,%rax
004717b0 <+176>: jne    0x4717a3 <main()+163>
```

deque
1.50s

```
004719a5 <+389>: movslq (%r9),%rcx
004719a8 <+392>: add    $0x1,%rax
004719ac <+396>: add    $0x4,%r8
004719b0 <+400>: sub    $0x1,%r10
004719b4 <+404>: add    %rcx,%rdx
004719b7 <+407>: cmp    %r11,%rax
004719ba <+410>: je     0x4719e0 <main()+448>
004719bc <+412>: cmp    $0x7f,%rax
004719c0 <+416>: mov    %r8,%r9
004719c3 <+419>: jbe    0x4719a5 <main()+389>
```

priority_queue<T>

- Implementation of a max heap
- Included in the **<queue>** header, but is not related to queue at all
- An adapter of **vector**
- **push(x)**: inserts **x**
- **top()**: returns the greatest element
- **pop()**: removes the greatest element

```
priority_queue<int> pq;
pq.push(1);
pq.push(3);
pq.push(2);
cout << pq.top() << endl; // 3
pq.pop();
cout << pq.top() << endl; // 2
```

priority_queue<T>

- If you need a min-heap
  - Method 1: negate all numbers

```
priority_queue<int> pq;
pq.push(-2);
pq.push(-1);
pq.push(-3);
int actual = -pq.top();
cout << actual << endl; // 1
```

  - Method 2: Specify **`greater<T>`** as the custom comparison function
    - Note: defined in `<functional>` header

```
priority_queue<string, vector<string>, greater<string>> pq;
pq.push("Grape");
pq.push("Apple");
pq.push("Banana");
cout << pq.top() << endl; // Apple
```

  - Method 3: Implement **`operator<`** with reversed logic (not recommended)

```
struct Student {
  string name;
  int height;
  bool operator<(const Student& o) const {
    return height > o.height;
  }
};
...
  priority_queue<Student> pq;
  pq.push(Student{"Terence", 200});
  pq.push(Student{"Hayden", 150});
  cout << pq.top().height << endl; // 150
```

Associative Containers

- Elements in an array / a vector are indexed by consecutive non-negative integers 0, 1, 2, ..., n - 1
- Associative containers allow element access based on keys
- The type of the key is not necessarily integral, but must implement strict weak ordering (e.g. **`operator<`**)
- Element access by key is $O(\lg n)$
- Associative containers are usually implemented as Red-Black trees

set<T>

- set is like an "always sorted" array

insert / erase
$O(\lg n)$

```
set<int> st;
st.insert(1000);
st.insert(12345);
st.insert(40);
cout << *st.begin() << endl;  // 40
cout << *st.rbegin() << endl; // 12345
st.insert(8);
st.erase(12345);
cout << *st.begin() << endl;  // 8
cout << *st.rbegin() << endl; // 1000
```

Initialize with unsorted sequence:
$O(n \lg n)$

```
set<string> st{"Banana", "Grape", "Apple"};
cout << *st.begin() << endl;  // Apple
cout << *st.rbegin() << endl; // Grape
```

Initialize with sorted sequence:
$O(n)$

```
set<double> st{1.0, 2.5, 3.6, 9.25};
cout << *st.begin() << endl;  // 1
cout << *st.rbegin() << endl; // 9.25
```

set<T>

$O(\lg n)$

- Two keys x and y are considered equal if (x < y) = (y < x) = false
- **`st.insert(x)`** returns **`pair<set<T>::iterator, bool>`**

iterator to the inserted/existing element

whether an new element is actually inserted

```
cout << st.insert(5).second << endl; // 1
cout << st.insert(6).second << endl; // 1
cout << st.insert(5).second << endl; // 0
cout << st.size() << endl;           // 2
cout << st.count(5) << endl;         // 1
st.erase(5);
cout << st.count(5) << endl;         // 0
```

erase using key

```
struct A {
  string s;
  bool operator<(const A& o) const {
    return s[0] < o.s[0];
  }
};
```

```
set<A> st;
st.insert(A{"Apple"});
st.insert(A{"Banana"});
cout << st.count(A{"Alice"}) << endl; // 1
auto ret = st.insert(A{"Alice"});
cout << ret.first->s << endl;       // Apple
cout << ret.second << endl;         // 0
cout << st.begin()->s << endl;      // Apple
st.erase(A{"Bob"});
cout << st.count(A{"Banana"}) << endl; // 0
```

set<T>

- **`set<T>::iterator`** is a BidirectionalIterator
- We can traverse a set by advancing and backing an iterator

```
set<int> st{1, 3, 5, 7, 9, 11};
auto it = st.begin();
++it;
++it;
cout << *it << endl; // 5
it = st.erase(it);
cout << *it << endl; // 7
--it;
cout << *it << endl; // 3
cout << distance(st.begin(), it) << endl; // 1
```

Erase using iterator

Both erase(x) and erase(it) version returns an iterator pointing to the element after the deleted one

Note: **`distance(it1, it2)`** is $O(it2 - it1)$ for BidirectionalIterator

set<T>

$O(\lg n)$

- Since set can be considered as an "always sorted" array (binary search tree actually), we can perform binary search on it
- **`lower_bound`** and **`upper_bound`** can also be used

```
vector<int> v{1, 3, 5, 7, 9, 11};
set<int> st(v.begin(), v.end());
// O(n) because the range is sorted
int x;
cin >> x;
auto it = st.lower_bound(x);
if (it != st.end()) {
  // prints the smallest key >= x
  cout << *it << endl;
} else {
  // x is greater than every key
  cout << "None" << endl;
}
```

Input: 3 Output: 3
Input: 6 Output: 7
Input: 12 Output: None

Warning:
`lower_bound(st.begin(), st.end(), x)`
will compile but is O(n)

map<K, M>

- Sometimes it might be more useful to store extra information other than the key
- **`map<K, M>`** is similar to **`set<K>`** but extra information of type **`M`** is added to each key

```
map<int, string> numberToName;
numberToName[23382338] = "McDonald's";
numberToName[21800000] = "KFC";
numberToName[23300000] = "Pizza Hut";
map<string, int> nameToNumber;
nameToNumber["McDonald's"] = 23382338;
nameToNumber["KFC"] = 21800000;
nameToNumber["Pizza Hut"] = 31800000;
nameToNumber["Pizza Hut"] = 23300000;
```

If key is not found,
an insertion is performed

If key is found,
the value is updated

map<K, M>

- **`mp.find(k)`** returns an iterator to the element with key **`k`** if it exists, returns **`mp.end()`** instead of key **`k`** does not exist
- The iterator points to the *value type*, which is a **`pair<const K, M>`** with **`first`** being the key and **`second`** being the mapped value

```
map<string, double> prices;
prices["USD"] = 7.8;
prices["GBP"] = 10.4;
auto it = prices.find("JPY");
cout << (it == prices.end()) << endl; // 1
it = prices.find("GBP");
cout << (it == prices.end()) << endl; // 0
cout << it->first << endl;            // GBP
it->second = 9.7;
cout << prices["GBP"] << endl;        // 9.7
```

Changing mapped value is $O(1)$ using iterator

```
cout << it->first << endl;
it->first = "EUR";
```

Compilation error:
Key cannot be changed this way

unordered_set / unordered_map

Access
Average: $O(1)$
Worst: $O(n)$

- Hash table implementation
- Since elements are unordered, you cannot perform binary search
- Also, begin() does not return the smallest element
- Handles collisions using *separate chaining*
- By default, when the number of elements (size) reaches the number of buckets (load factor = 1.0), the capacity increases (~2x) and all elements are rehashed

```
unordered_set<int> us;
for (int i = 1; i <= 100; i++) {
  us.insert(i);
  cout << i << ' ' << us.bucket_count() << endl;
}
```

Output:

```
1  11      22  23
2  11      23  47
...        ...
10 11      46  47
11 23      47  97
```

unordered_set / unordered_map

- In order to use **`unordered_set/map`**, classes must have hash function implemented
- Most compilers have already implemented hash functions for common types such as **`int, double, string`**, but not for **`pair`**

Hasher class implements operator()

```
struct Hasher {
  size_t operator()(pair<int, int> x) const {
    return x.first ^ x.second;
  }
};
......
  unordered_set<pair<int, int>, Hasher> us;
  us.insert({4, 2});
  us.insert({0, 6});
  us.insert({10, 12});
  cout << us.bucket_size(6) << endl; // 3
```

specify hasher class

0	1	2	3	4	5	6	7	8	9	10
						{4, 2} {0, 6} {10, 12}				

unordered_set / unordered_map

- However, since element access is O(n) worst case, it is possible to craft test cases that causes most element access to take O(n) time
- Also, the hash function of integers is weak

```
cout << hash<int>{}(123) << endl; // 123
cout << hash<int>{}(124) << endl; // 124
```

- This is especially true if people can look at your code (e.g. Codeforces) http://codeforces.com/blog/entry/44731
- In such cases, use **`set/map`** instead of **`unordered_set/map`**
- Alternatively, implement a stronger hash function that produces ***different, unpredicatble*** hashes for the same number ***across runs***

(unordered_)multiset

- For **(unordered_)set**, there is a **multi** version which allow duplicate keys

```
unordered_multiset<int> ums;
ums.insert(123);
ums.insert(123);
cout << ums.count(123) << endl; // 2
```

- However, the time complexity of count is additionally linear to the number of elements having the key

```
unordered_multiset<int> ums;
long long sum = 0;
for (int i = 1; i <= 10000; i++) {
  ums.insert(1);
  sum += ums.count(1);
}
cout << sum << endl; // 50005000
```

0.48s

(unordered_)multimap

- Elements can no longer be accessed using [ ]

```
multimap<int, int> mmp;
for (int i = 1; i <= 10000; i++) {
  mmp[1] = i;
}
```

- Therefore there are very few use cases

Frequency map

- Since multiset count could be slow, we can use map to store the frequency of a key instead of inserting multiple copies of they same key into a multiset

```
unordered_map<int, int> ump;
long long sum = 0;
for (int i = 1; i <= 10000; i++) {
  ump[1]++;
  sum += ump[1];
}
cout << sum << endl; // 50005000
```

0.01s

2 2 2 3 3 5 7 7

Key	2	3	5	7
Mapped Value	3	2	1	2

Frequency map

- Implement a frequency map that supports:
  - Insert a number x
  - Delete a number x
  - Find the current size
  - Find the smallest number
  - Find the largest number
  - Find the number of 'x's
  - Find smallest element > x

Frequency map

- Let's assume that we never remove numbers from the frequency map first
- To insert a number x

```
map<int, int> freq;
int size = 0;

freq[x]++;
size++;
```

Note: when you access key x using [ ], and if it does not exist, an element with key x value 0 is automatically inserted

- Number of 'x's : `freq[x]`

2 2 2 3 3 5 7 7

Key	2	3	5	7
Mapped Value	3	2	1	2

2 2 2 3 3 3 5 7 7

Key	2	3	5	7
Mapped Value	3	**3**	1	2

Frequency map

```
map<int, int> freq{
  {2, 3}, {3, 3}, {5, 1}, {7, 2},
};
```

- Find the smallest number

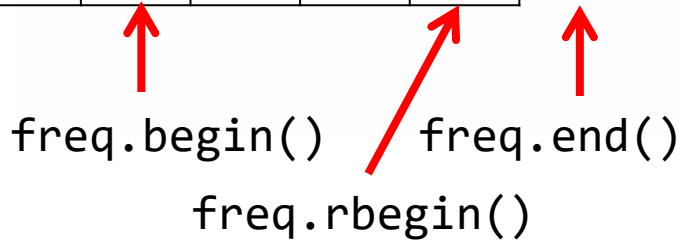
```
cout << freq.begin()->first << endl; // 2
```

- Find the largest number

```
cout << freq.rbegin()->first << endl; // 7
```

2 2 2 3 3 3 5 7 7

Key (first)	2	3	5	7
Mapped Value (second)	3	3	1	2



Frequency map

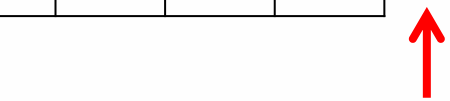
- We can use **upper_bound** to find the smallest number > x

```
map<int, int> freq{
  {2, 3}, {3, 3}, {5, 1}, {7, 2},
};

cout << freq.upper_bound(0)->first << endl;          // 2
cout << freq.upper_bound(2)->first << endl;          // 3
cout << freq.upper_bound(3)->first << endl;          // 5

cout << freq.upper_bound(10) == freq.end() << endl; // 1
// possible runtime error
cout << freq.upper_bound(10)->first << endl;
```

Key (first)	2	3	5	7
Mapped Value (second)	3	3	1	2



`freq.upper_bound(10) == freq.end()`

Frequency map

- To remove a number x we need to subtract its frequency by 1

```
freq[x]--;
```

- Also, we need to remove the key if there is no 'x's left

```
if (freq[x] == 0) {
  freq.erase(x);
}
```

- Faster:

```
auto it = freq.find(x);
if (--it->second == 0) {
  freq.erase(it);
}
```

Key (first)	2	3	5	7
Mapped Value (second)	3	3	1	2

↓ remove 7

Key (first)	2	3	5	7
Mapped Value (second)	3	3	1	1

↓ remove 7

Key (first)	2	3	5
Mapped Value (second)	3	3	1

bitset<L>

- Creates a container for storing L 0/1 (boolean) values
- The length L must be specified at declaration and is fixed

```
int n;
cin >> n;
bitset<10000000> bs;
bs[0] = 1;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  bs |= bs << x;
}
for (int i = 0; i < 10000000; i++) {
  if (bs[i]) {
    cout << i << ' ';
  }
}
```

```
cout << sizeof(bitset<1000>) << endl; // 128
```

Input:
4
3 4 9 25
Output:
0 3 4 7 9 12 13 16 25 28 29 32 34 37 38 41

bitset<L>

```
int n = 1000;
vector<bool> b(10000001);
b[0] = 1;
for (int i = 0; i < n; i++) {
  int x = i * 4 + 1987;
  for (int j = 10000000; j >= x; j--) {
    b[j] = b[j] | b[j - x];
  }
}
int sum = 0;
for (bool x : b) {
  sum += x;
}
cout << sum << endl; // 3973049
```

20.8s

```
int n = 1000;
bitset<10000001> bs;
bs[0] = 1;
for (int i = 0; i < n; i++) {
  int x = i * 4 + 1987;
  bs |= bs << x;
}
cout << bs.count() << endl; // 3973049
```

0.89s

```
int n = 1000;
vector<unsigned long long> a(156251);
a[0] = 1ull;
for (int i = 0; i < n; i++) {
  int x = i * 4 + 1987;
  int indices = x >> 6;
  int bits = x & 63;
  int bits2 = 64 - bits;
  for (int i = 156250; i > indices; i--) {
    a[i] |= (a[i - indices] << bits);
    a[i] |= (a[i - indices - 1] >> bits2);
  }
  a[indices] |= a[0] << bits;
}
int sum = 0;
for (int i = 0; i <= 156250; i++) {
  sum += __builtin_popcountll(a[i]);
}
cout << sum << endl; // 3973049
```

0.20s

Iterator validity

Note: simplified Read the docs for details

- Depending on the container, iterators may be invalidated after an element is inserted/erased
- `vector`
  - `push_back`: All iterators are invalidated if a reallocation occurs
- `deque`
  - Insert / erase: All iterators are invalidated
- `list`
  - Insert: All other validators remain valid
  - Erase: Iterators to erased element(s) are invalidated

Iterator validity

Note: simplified Read the docs for details

- `set` / `map`
  - Insert: All validators remain valid
  - Erase: Iterators to erased element(s) are invalidated

```
set<int> st{1, 2, 3};
auto it = st.begin();
st.erase(it); // {2, 3}
cout << *next(it) << endl; // possible runtime error
```

```
set<int> st{1, 2, 3};
auto it = st.begin();
it = st.erase(it); // {2, 3}
cout << *it << endl; // 2
```

- `unordered_set` / `unordered_map`
  - Insert: All iterators are invalidated if a rehash occurs
  - Erase: Iterators to erased element(s) are invalidated

Iterator validity

- For all containers, iterators pointing to **`.end()`** will always become invalid after a size change

```
vector<int> v{1, 2, 3, 4, 5, 6};
auto it = v.end();
v.pop_back();
v.pop_back();
v.pop_back();
cout << *prev(it) << endl;
```

shared_ptr

- Problem with raw pointers

```
int* a = new int[10];
cout << a << endl;
a[1] = 5;
delete [] a;
cout << a << endl;
a[1] = 10; // runtime error
```

Possible output:
0x16969d0
0x16969d0

- **shared_ptr** uses reference counting to manage object lifecycle

```
shared_ptr<string> ptr = make_shared<string>("abcdef");
cout << ptr->length() << endl; // 6
```

Class Constructor arguments

shared_ptr

```
struct TrieNode {
  int count;
  vector<shared_ptr<TrieNode>> a;
  TrieNode(): count(0), a(26) {}
};
shared_ptr<TrieNode> traverse(shared_ptr<TrieNode> root, const string& s) {
  shared_ptr<TrieNode> cur = root;
  for (char c : s) {
    if (!cur->a[c - 'a']) {
      cur->a[c - 'a'] = make_shared<TrieNode>();
    }
    cur = cur->a[c - 'a'];
  }
  return cur;
}
```

```
int main() {
  shared_ptr<TrieNode> root = make_shared<TrieNode>();
  traverse(root, "a")->count++;
  traverse(root, "abc")->count++;
  traverse(root, "abd")->count++;
  traverse(root, "abc")->count++;
  cout << traverse(root, "abc")->count << endl; // 2
  cout << traverse(root, "a")->count << endl;   // 1
  cout << traverse(root, "abe")->count << endl; // 0
  return 0;
}
```