Advanced C++

Anson Ho

27 April 2019

Objectives

- Shorter code
  - Given constant accuracy, shorter code implies less bugs
- More efficient code
  - Unfortunately, most optimizations here won't improve time complexity
- Motivate you to learn more
  - Actually, the topics covered in this sessions are not "advanced" among all C++ features
- Unlike other sessions, this session will focus on implementation more (in a specific language :P)

C++11

- Some topics in this session require `C++11`
- g++ flag: `-std=c++11`
  - You may use `export CXXFLAGS='-std=c++11'` if you are using `make` to compile in Linux
- Support: HKOI Online Judge, IOI, ...
- Not support: NOI :(
- The newer standards (`C++14`, `C++17`) are backward compatible in most cases

C++ Standard Library

- Many functions and classes that you may use directly after including suitable headers
- Standard Template Library (STL) is a subset of it
  (e.g. vector, queue, map, sort)
- Also include everything from C Standard Library (e.g. math functions)
- Most of them are declared in the namespace `std`
- Shortcuts
  - `#include <bits/stdc++.h>`
    - Not a standard header file
  - `using namespace std;`

Template

```
template<class T>
T square(const T a) {
  return a * a;
}

int main() {
  auto a = square<int>(42) +
           square<short>(42) +
           square<double>(42) +
           square<float>(42) +
           square(42);
}
```

Template

- Examples: class template, function template
- Template parameters: usually a generic type (e.g. `class T`)
- Usually, you have to specify the target type when you are referring to the template
- Compiler will generate a copy of the target class / function for each used type with corresponding substitution
- In most cases, you use templates instead of writing templates
  - Examples: `vector<int>`, `queue<long long>`, `complex<double>`

Template

```
g++ -pg -std=c++11 template.cpp -o template
./template
gprof -b template | head -n 11 > result.txt
```

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        2     0.00     0.00  int square<int>(int)
  0.00      0.00     0.00        1     0.00     0.00  double square<double>(double)
  0.00      0.00     0.00        1     0.00     0.00  float square<float>(float)
  0.00      0.00     0.00        1     0.00     0.00  short square<short>(short)
```

std::pair and std::tuple

- Defined in `<utility>` and `<tuple>` respectively
- `std::pair` is a struct template to store two objects (possibly of different types)
  `std::tuple` is a generalization of `std::pair` ($2 \rightarrow$ N)
- Data members of `pair<int, int> a` can be accessed by `a.first` and `a.second`
  Data members of `tuple<int, int, int> b` can be accessed by `get<0>(b)`, `get<1>(b)` and `get<0>(b)`
  - `N` in `get<N>(b)` must be known in compile time
  - Use array or `std::vector` if not

std::pair and std::tuple

- Comparison (<, <=, ==, ...) works with lexicographical order if all types are comparable
- It is possible that `sizeof(pair<T1, T2>)` $>$ `sizeof(T1)` $+$ `sizeof(T2)`
  - It may also happen in struct and class
  - You can disable the padding but it makes things slower

std::pair and std::tuple

```
int main() {
  pair<int, long long> a(10, 12);
  pair<int, long long> b = {10, 11};
  assert(b < a);
  assert(a.first + 1 == b.second);
  pair<int, int> c = make_pair(13, 14);
  tie(a.first, b.first) = c;
  assert(a.first == 13 && b.first == 14);
  tuple<int, int, int> d = make_tuple(15, 16, 17);
  get<0>(d) = get<1>(d) + get<2>(d);
  assert(get<0>(d) == 16 + 17);
}
```

std::vector

- Defined in `<vector>`
- A container of a type specified by user
  - Examples: `vector<int>`, `vector<MyStruct>`, `vector<vector<int>>`
- Similar to an array (O(1) random access), but with dynamic size
- You can push / pop elements at the end of the `std::vector`, like stack
- Push N elements
  - Time complexity: O(N)
  - Space complexity: O(N)

std::vector

- Comparison (<, <=, ==, ...) works with lexicographical order if the specified type is comparable
- 2D vector
  - `int x[n][m];`
  - `vector<vector<int>> x(n, vector<int>(m);`
- Practical example: storing edges of a graph
  - adjacency matrix: O(N^2) memory
  - linked list: jumping address

std::vector

```
int main() {
  vector<int> a(5);
  vector<int> b(5, 0);
  vector<int> c({0, 0, 0, 0, 0});
  vector<int> d = {0, 0, 0, 0, 0};
  vector<int> e;
  for (int i = 0; i < 5; i++)
    e.push_back(0);
  assert(a == b && b == c && c == d && d == e);
}
```

std::vector

```
int main() {
  vector<int> x(5);
  for (int i = 0; i < x.size(); i++)
    x[i] += 1;
  for (int i = (int)x.size() - 1; i >= 0; i--)
    x[i] += 10;
  for (auto i : x)
    i += 100;
  for (auto& i : x)
    i += 1000;
  for (int i = 0; i < x.size(); i++)
    assert(x[i] == 1011);
}
```

std::vector

Behind the scenes: reallocation

- Data are stored in an internal ”array” (allocated memory)
- When the ”array” is not large enough, a larger ”array” will be ”created” (usually with double size)
- Everything in the old ”array” will then be moved to the new ”array”
- ”Array” size $\leq 2N$
- Number of moves $\leq 1 + 2 + 4 + \cdots + 2^{\lceil \log N \rceil} < 4N$

std::vector

```
int main() {
  vector<int> x;
  for (int i = 0; i < 10; i++) {
    printf("size=%d capacity=%d\n",
           (int)x.size(),
           (int)x.capacity());
    x.push_back(i);
  }
}
```

```
size=0 capacity=0
size=1 capacity=1
size=2 capacity=2
size=3 capacity=4
size=4 capacity=4
size=5 capacity=8
size=6 capacity=8
size=7 capacity=8
size=8 capacity=8
size=9 capacity=16
```

std::vector

- `emplace_back` vs `push_back`
  - shorter and faster, especially for large struct
- Iterators and references are invalid after reallocation
- `swap` (member function) vs `std::swap`
  - $O(1)$ vs $O(N)$
- `x.clear()` vs `vector<int>().swap(x)`
- Use `reserve` or `resize` if you know the size in advance

std::deque

- Defined in `<deque>`
- A double-ended queue with random access
- Differences between `std::deque` and `std::vector` (actually very similar)
  - `std::deque` has `push_front` and `pop_front`
  - Memory usage
    - `std::deque`: $N + c_1$
    - `std::vector`: $2N + c_2$
    - $c_1 > c_2$
    - Be careful of the memory used by many `std::deque`s of small size
  - `std::deque` never invalidates iterators and references
  - The internal storage of `std::deque` are not contiguous

std::deque

```
int main() {
  deque<int> q;
  q.push_back(1);
  q.push_back(2);
  q.push_front(3);
  q[1] += 100;
  q.pop_front();
  q.pop_back();
  assert(q.front() == 101);
}
```

std::deque

- g++ implementation of `std::deque`
  - Allocate a chunk of memory of fixed size when necessary
    - `#define _GLIBCXX_DEQUE_BUF_SIZE 512`
  - The pointers to each chunk of memory are stored in a container similar to `std::vector`
- It is possible to implement `std::deque` yourself by a circular buffer with `std::vector` as container

std::deque

```
int main() {
  int n;
  scanf("%d", &n);
  deque<int> q[10000];
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < n; j++) {
      q[i].push_back(j);
    }
  assert(q[9999].size() == n);
}
```

```
n=1     Maximum resident set size (kbytes): 10088
n=2     Maximum resident set size (kbytes): 10232
n=3     Maximum resident set size (kbytes): 10112
n=4     Maximum resident set size (kbytes): 10092
n=5     Maximum resident set size (kbytes): 10216
n=123   Maximum resident set size (kbytes): 10212
n=124   Maximum resident set size (kbytes): 10080
n=125   Maximum resident set size (kbytes): 10092
n=126   Maximum resident set size (kbytes): 10128
n=127   Maximum resident set size (kbytes): 10120
n=128   Maximum resident set size (kbytes): 15256
n=129   Maximum resident set size (kbytes): 15236
n=130   Maximum resident set size (kbytes): 15388
n=131   Maximum resident set size (kbytes): 15180
n=132   Maximum resident set size (kbytes): 15236
n=133   Maximum resident set size (kbytes): 15228
n=255   Maximum resident set size (kbytes): 15332
n=256   Maximum resident set size (kbytes): 20488
n=257   Maximum resident set size (kbytes): 20336
```

std::vector

```
int main() {
  int n;
  scanf("%d", &n);
  vector<int> q[10000];
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < n; j++) {
      q[i].push_back(j);
    }
  assert(q[9999].size() == n);
}
```

```
n=1     Maximum resident set size (kbytes): 3912
n=2     Maximum resident set size (kbytes): 3944
n=3     Maximum resident set size (kbytes): 4040
n=4     Maximum resident set size (kbytes): 3948
n=5     Maximum resident set size (kbytes): 4136
n=123   Maximum resident set size (kbytes): 8824
n=124   Maximum resident set size (kbytes): 8760
n=125   Maximum resident set size (kbytes): 8776
n=126   Maximum resident set size (kbytes): 8744
n=127   Maximum resident set size (kbytes): 8888
n=128   Maximum resident set size (kbytes): 8792
n=129   Maximum resident set size (kbytes): 13888
n=130   Maximum resident set size (kbytes): 13884
n=131   Maximum resident set size (kbytes): 13696
n=132   Maximum resident set size (kbytes): 13844
n=133   Maximum resident set size (kbytes): 13752
n=255   Maximum resident set size (kbytes): 13888
n=256   Maximum resident set size (kbytes): 13772
n=257   Maximum resident set size (kbytes): 23560
```

std::queue and std::stack

- Defined in `<queue>` and `<stack>` respectively
- Container adapters for FIFO and LIFO data structures respectively
  See Data Structures (I)
- No random access
- `std::deque` is the default container, which can be changed
  - `pop_front` in `std::deque` becomes `pop` in `std::queue`
  - `push_back` in `std::deque` becomes `push` in `std::stack`

std::queue and std::stack

```
int main() {
  queue<int> q;
  q.push(1);
  q.push(2);
  assert(q.front() == 1);
  q.pop();
  assert(q.front() == 2);
}
```

```
int main() {
  stack<int> s;
  s.push(1);
  s.push(2);
  assert(s.top() == 2);
  s.pop();
  assert(s.top() == 1);
}
```

std::priority_queue

- Defined in `<queue>`
- Similar to `std::queue` but the front (top) of the queue becomes the greatest element (defined by the operator $<$)
  See binary heap in Data Structures (II)
- `std::vector` is the default container, which can be changed

std::priority_queue

```
int main() {
  priority_queue<int> q;
  q.push(1);
  q.push(4);
  q.push(2);
  q.push(3);
  assert(q.top() == 4);
  q.pop();
  assert(q.top() == 3);
}
```

std::priority_queue

```
int main() {
  priority_queue<int, vector<int>, greater<int>> q;
  q.push(1);
  q.push(4);
  q.push(2);
  q.push(3);
  assert(q.top() == 1);
  q.pop();
  assert(q.top() == 2);
}
```

std::priority_queue

```
struct Cmp {
  bool operator()(int a, int b) {
    return a > b;
  }
};

int main() {
  priority_queue<int, vector<int>, Cmp> q;
  q.push(1);
  q.push(4);
  q.push(2);
  q.push(3);
  assert(q.top() == 1);
  q.pop();
  assert(q.top() == 2);
}
```

std::priority_queue

- No write access to the elements
  but you can pop → edit → push
- No builtin comparison (of the whole object)
- HKOJ 30107

std::set and std::map

- Defined in `<set>` and `<map>` respectively
- Associative containers
- Unique key with partial ordering (defined by operator $<$)
- Every key is associated with a value in `std::map`
  which can be accessed by operator `[]`
- Time complexity: O($\log N$) for each operation
  Space complexity: O($N \log N$)
- g++ implementation: red-black tree

std::set

```
int main() {
  set<int> s;
  s.insert(1);
  s.insert(1);
  s.insert(2);
  assert(s == set<int>({1, 2}));
  for (int i : s)
    printf("%d ", i);
  // 1 2
  s.erase(1);
  assert(s == set<int>({2}));
}
```

std::map

```
int main() {
  map<int, int> m;
  m[1] = 2;
  m.insert({1, 3});
  m[4] = 5;
  m[4] = 6;
  for (auto& i : m)
    printf("%d,%d ", i.first, i.second);
  // 1,2 4,6
}
```

std::set and std::map

- If all updates happen before queries, `std::vector` is a faster option
- Comparison (<, <=, ==, ...) works with lexicographical order
  - `set<map<vector<int>, int>>` is a valid type
- Using `insert` and play with the iterator instead of `[]` can increase performance sometimes
  - `insert` returns `pair<iterator, bool>`

std::multiset and std::multimap

- Defined in `<set>` and `<map>` respectively
- Very similar to `std::set` and `std::map`
- Key is no longer unique
- Be careful when using `erase`
- Use `std::map` instead of `std::multiset` and `count`

std::unordered_set and std::unordered_map

- Defined in `<unordered_set>` and `<unordered_map>` respectively
- Operator $<$ is no longer required
  - Iterator traversal don't depends on the order of key
- A hash is required
  - Built-in hash for int, long long, ...
- Implementation: hash table
  See Data Structure (II)

std::unordered_set

```
struct Hash {
  size_t operator()(const pair<int, int>& a) const {
    return a.first ^ a.second;
  }
};

int main() {
  unordered_set<pair<int, int>, Hash> s;
}
```

std::unordered_set and std::unordered_map

- Time complexity: expected $O(1)$ for each operation
  Worst case: $O(N)$
  - Hackers (and problem setters) can easily let you get TLE if you use built-in hash simply
- Use `reserve` and `max_load_factor`
- `std::unordered_multiset` and `std::unordered_multimap`

std::sort

- Defined in `<algorithm>`
- Worst case time complexity: $O(N \log N)$
  - `qsort` in `<cstdlib>` has worst case time complexity $O(N^2)$
- The range parameters must be random access iterators
  - Examples: pointers, iterators of `std::vector`
  - Non-examples: iterators of `std::list`
- `std::stable_sort` is slightly slower
  but preserves order between equal elements

std::sort

Optionally, a custom comparison function can be used.

- The function must implement a strict partial order $<$
  - Irreflexivity
    - $a < a$ is always false
  - Asymmetry
    - If $a < b$ is true, then $b < a$ is false
  - Transitivity
    - If $a < b$ and $b < c$ are true, then $a < c$ is true
- Example: use one data member as key, another as tiebreaker
- Tiebreaker is not a requirement, similar to topological order in Graph (III)

std::sort

```
bool cmp(int a, int b) {
  return to_string(a) < to_string(b);
}

int main() {
  int x[] = {1, 13, 100, 42};
  sort(x, x + 4);
  printf("%d %d %d %d\n", x[0], x[1], x[2], x[3]);
  // 1 13 42 100
  sort(x, x + 4, cmp);
  printf("%d %d %d %d\n", x[0], x[1], x[2], x[3]);
  // 1 100 13 42
}
```

std::sort

```
struct Pair {
  int first, second;
} x[5];

int main() {
  // compilation error
  sort(x, x + 5);
}
```

std::sort

```
struct Pair {
  int first, second;
  bool operator<(const Pair& rhs) const {
    if (first != rhs.first)
      return first < rhs.first;
    return second < rhs.second;
  }
} x[5];

int main() {
  sort(x, x + 5);
}
```

std::sort

```
struct Pair {
  int first, second;
} x[5];

bool operator<(const Pair& a, const Pair& b) {
  if (a.first != b.first)
    return a.first < b.first;
  return a.second < b.second;
}

int main() {
  sort(x, x + 5);
}
```

Binary search in <algorithm>

- `std::lower_bound`, `std::upper_bound`, `std::binary_search`
- Required to be sorted in advance
- Use `lower_bound` (and `upper_bound`) in `std::set` (and `std::map`)
- Be careful of the template issue
  `https://codeforces.com/blog/entry/66287`

Binary search in <algorithm>

```
int main() {
  vector<int> v({1, 2, 3, 3, 5});
  assert(lower_bound(v.begin(), v.end(), 3) == v.begin() + 2);
  assert(upper_bound(v.begin(), v.end(), 3) == v.begin() + 4);
  assert(*upper_bound(v.begin(), v.end(), 3) == 5);
  assert(lower_bound(v.begin(), v.end(), 6) == v.end());
  assert(upper_bound(v.begin(), v.end(), 5) == v.end());
  assert(binary_search(v.begin(), v.end(), 2) == true);
  assert(binary_search(v.begin(), v.end(), 4) == false);
}
```

More in <algorithm>

- `std::unique`
- `std::swap`
- `std::fill`
- `std::reverse`
- `std::next_permutation`
- `http://www.cplusplus.com/reference/algorithm/`

std::bitset

- Defined in `<bitset>`
- A memory efficient storage of bits of fixed number
- Support bitwise operation (`|`, `^`, `&`, `|=`, ...)
  - Time complexity: $O(\frac{N}{32})$ or $O(\frac{N}{64})$
- Sometimes useful in solving problems (in a non-intended way)
  - `https://csacademy.com/contest/fii-code-2019-round-1/task/Sugarel-s-Garden/`
- Some non-standard member functions
  - `_Find_first`, `_Find_next`
  - `https://codeforces.com/blog/entry/43718`

More in C++ Standard Library

- `<string>`
  - Different from C-styled string
- `<list>`
- `<chrono>`
- `<functional>`
  - `std::sort`
  - `std::all_of`, `std::any_of`
  - Write code sequentially
- `https://en.wikipedia.org/wiki/C++_Standard_Library`

Resources

- https://en.cppreference.com
- https://stackoverflow.com/questions/81656/where-do-i-find-the-current-c-or-c-standard-documents
- https://github.com/gcc-mirror/gcc/tree/master/libstdc++-v3/include

Non-standard library

- Possibly not existing in some C++ compilers
- usable in g++ (which is used in HKOI Online Judge)
- Lack of good (and official) documentation

Non-standard library

- https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/
- https://github.com/kth-competitive-programming/kactl/blob/master/content/data-structures/OrderStatisticTree.h
- Implicit cartesian tree in GNU C++ STL.
  https://codeforces.com/blog/entry/10355
- https://www.luogu.org/blog/Chanis/gnu-pbds

Non-standard library

- HKOJ AP121 M1913
- Luogu P3369
- BZOJ 1269
- `https://www.hackerrank.com/contests/zalando-codesprint/challenges/give-me-the-order`