Нормальный C++ за 2 пары

Сапожников Денис

Содержание

1 Введение в C++

C++ - это очень мощный язык программирования, его можно изучать и постигать долгие годы и оказывается, что он очень крутой, но почему-то многие так не считают. Давайте я вам расскажу, что умеет делать современный C++ в приложении к олимпиадному программированию.

1.1 Классы

```cpp
template <class T>
struct Vect {
  T x, y;
  Vect(T x = 0, T y = 0): x(x), y(y) {}
  Vect(Vect a, Vect b): x(b.x - a.x), y(b.y - a.y) {}
  Vect operator+(const Vect other) {
    return { x + other.x, y + other.y };
  }
  Vect operator-(const Vect other) {
    return { other.x - x, other.y - y };
  }
  Vect operator+=(const Vect other) {
    return *this += other;
  }
  Vect operator-=(const Vect other) {
    return *this -= other;
  }
  Vect operator*(T k) {
    return { x * k, y * k };
  }
  T operator*(const Vect other) {
    return x * other.x + y * other.y;
  }
  friend istream &operator>>(istream &in, Vect &v) {
    return in >> v.x >> v.y;
  }
  friend ostream &operator<<(ostream &out, const Vect &v) {
    return out << v.x << ' ' << v.y;
  }
};
```

1.2 Сортировка, лямбда-функции

Предположим, вы хотите что-нибудь отсортировать, но вам нужно сортировать свои структуры или встроенные, но не по возрастанию, а как-нибудь по-другому, как это можно сделать?

Разберёмся на конкретном примере:

Пусть у нас есть класс Student, мы хотим отсортировать всех школьников сначала по убыванию средней оценки, а при равенстве средней оценки лексикографически сначала по фамилии, затем по имени.

```
struct Student {
  double mark;
  string surname, name;
};
```

Функция, которая говорит, что студент a меньше студента b (то есть должен идти раньше в отсортированном массиве) называется компаратор. Из её описания логично, что она возвращает булевское значение. Конкретно в нашем примере компаратор будет такой:

```
bool cmp(const Student &a, const &Student b) {
  return a.mark > b.mark ||
       a.mark == b.mark && a.surname < b.surname ||
       a.mark == b.mark && a.surname == b.surname && a.name < b.name;
}
```

Тогда чтобы отсортировать по нашему компаратору вектор, достаточно написать:

```
sort(students.begin(), students.end(), cmp);
```

Вам не кажется, что компаратор, который написан непонятно где сверху в программе немного может запутать? Оказывается, существуют лямбда-функции - это функции, описанные прямо в месте их вызова. У них следующий синтаксис:

```
auto f = [global variable](parameters) {
  // code
}
```

Вместо global variable вы пишете переменные, которые должны быть видны внутри функции, важно передавать их **по ссылке**, иначе они каждый раз будут копироваться при вызове лямбда-функции. Ещё можно написать только & и получить доступ сразу ко всем переменным, не перечисляя каждую.

Как использовать это в нашей сортировке?

```
sort(students.begin(), students.end(), [](const Student &a, const
    Student &b) {
  return a.mark > b.mark ||
       a.mark == b.mark && a.surname < b.surname ||
       a.mark == b.mark && a.surname == b.surname && a.name < b.name;
});
```

Кажется, так более читабельно и на самом деле чуть быстрее работает.

Иногда нам нужен, например, std::set<Student>. Чтобы он автоматически сортировал по этому компаратору, нужно определить operator< у нашей структуры. **Важно делать его константным**, иначе вы будете получать неведомую ошибку и проклинать мир на олимпиаде.

```cpp
struct Student {
  //...
  bool operator<(const Student& other) const {
    return mark > other.mark ||
          mark == other.mark && surname < other.surname ||
          mark == other.mark && surname == other.surname && name < other
              .name;
  }
}

int main() {
  set<Student> students;
}
```

Замечание. Компаратор и operator< должны быть именно строго $<$, а не $\leq$, то есть эти функции должны задавать порядок на множестве, который удовлетворяет аксиомам строго порядка, а именно:

1. $\neg(a < a)$ — антирефлексивность
2. $a < b, b < c \Rightarrow a < c$ — транзитивность
3. $a < b \Rightarrow \neg(b < a)$ — асимметричность

А теперь немного другой пример: пусть у нас есть структура A с 4 полями интов, мы хотим сортировать сначала по возрастанию первого, потом по убыванию второго, потом по возрастанию третьего и четвёртого.

Чтобы не писать огромные конструкции, как мы делали это в прошлом примере, можно воспользоваться структурой std::tuple. Это кортеж переменных, при чём, возможно, разных типов, но в нашем - 4 инта. У него есть встроенная сортировка лексикографически по параметрам в том порядке, в котором мы её создадим. Чтобы создать *tuple*, мы можем воспользоваться *make_tuple*. Таким образом посортировать эту структуру мы можем так:

```cpp
struct A {
  int a, b, c, d;
  bool operator<(const A& other) const {
    return make_tuple(a, -b, c, d) < make_tuple(other.a, -other.b,
        other.c, other.d);
  }
}
```

Вот теперь уже это очень классно выглядит и приятно пишется!

1.3 Автоматическая типизация

Мы можем ещё немного улучить прошлый пример. В C++17 появилась автоматическая типизация, а именно в некоторых случаях вам не нужно писать, от каких типов мы созаем структуру, например vector/map/set/pair/tuple.

```
struct A {
  int a, b, c, d;
  bool operator<(const A& other) const {
    return tuple(a, -b, c, d) < tuple(other.a, -other.b, other.c, other
        .d);
  }
}
```

Или другой пример:

```
vector cpp17 = { 1, 2.1, 3 };
```

1.4 Пацанские циклы и Structural bindings

Идеальный пример для этого раздела — это взвешенный граф. Обычно вы его храните и бегаете по нему так:

```
const int N = 1e5;
vector<pair<int, int>> gr[N]; // { u, weight }

void dfs(int v) {
  used[v] = true;
  for (int i = 0; i < gr[v].size(); i++)
    if (!used[gr[i].first])
      dfs(gr[i].first);
}
```

Это просто ужас!

Давайте пробежимся немного получше с помощью цикла, который пробегает по любой коллекции (вектор, сет, мап, строка), появившийся в C++11 и называющийся foreach:

```
void dfs(int v) {
  used[v] = true;
  for (auto &edge : gr[v])
    if (!used[edge.first])
      dfs(edge.first);
}
```

Вот это уже было получше, вам не нужно страдать с индексами и т.д., но это всё ещё ужасно, потому что вы не должны помнить, что такое .first и .second у структуры. На помощь приходит C++ и Structural bindings:

```
void dfs(int v) {
  used[v] = true;
  for (auto &[u, w] : gr[v])
    if (!used[u])
      dfs(u);
}
```

Это же потрясающе выглядит и читаемость повышается на 100500%

1.5 Декомпозиция

Декомпозиция на самом деле это то же самое, что и Structural bindings. Допустим, у вас есть функция, которая возвращает пару — размер матрицы. До C++17 вам бы пришлось написать так:

```
auto sz = matrix_size(a);
n = sz.first, m = sz.second;
```

Но на помощь приходит C++17 и теперь код становится проще:

```
auto [n, m] = matrix_size(a);
```

1.6 Фишечки

Вы можете с ходу сказать, сколько нулей в числе?

```
const int N = 10000000;
```

Уверен, что нет, а, главное, в таком месте могут возникнуть неприятные баги. В C++17 теперь можно разделять знаки апострофом, то есть:

```
const int N = 10'000'000;
```

1.7 Читаем из файла

В C++ много вариантов, как можно читать из файла, самый популярный из них среди олимпиадных программистов – это такой:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Этот способ пришёл к нам из языка C и является устаревшим и небезопасным. Он перенаправляет стандартные потоки через файлы. Чтобы он скомпилировался, нужно до подключения библиотеки ввода-вывода написать следующее:

```
#define _CRT_SECURE_NO_WARNINGS
```

Но этот способ не подходит, когда нам нужно читать/выводить сразу из нескольких файлов, а такое бывает, например, на МОШе.

В таком случае можно сделать так:

```
#include <fstream>
#include <string>

using namesapce std;

int main() {
  for (int i = 0; i < 2; i++) {
    ifstream in(to_string(i) + ".txt");
    ofstream out("ans" + to_string(i) + ".txt");
```

```
    int x;
    in >> x;
    out << x + 1;
  }
}
```

Аналогично на самом деле можно переопределить локально внутри функции переменные *cin* и *cout*. Обратите внимание, что это будет работать только внутри main!

```
#include <fstream>
using namespace std;
int main() {
  ifstream cin("input.txt");
  ofstream cout("output.txt");
  int x;
  cin >> x;
  cout << x + 1;
}
```

1.8 Настраиваем вашу IDE

Если на вашем компе стоит Visual Studio 2014, то мне вас жаль. Чтобы можно было нормально запускать файлы без точек останова и т.д. проделываем такой путь:

Создаём новое консольное приложение → правой кнопкой мыши по проекту → свойства проекта → C/C++ → Предварительно откомпилированный заголовок → Не использовать

В настройках проекта удобно написать свой дефайн по типу *LOCAL*, чтобы можно было локально читать из файла, и, не меняя код, сразу засылать его в систему так, чтобы в системе он читал из стандартного потока ввода:

```
#ifdef LOCAL
  freopen("input.txt", "r", stdin);
#endif
```

Это делается по пути: свойства проекта → C/C++ → препроцессор → определения препроцессора.

Начиная с C++11 freopen считается небезопасным, и нужно прописать ещё и дефайн _CRT_SECURE_NO_WARNINGS, рекомендую это тоже сделать в ом же месте.

Проблема среды - вы не можете нормально внутри одного проекта работать с несколькими файлами. Чтобы не комментировать весь код каждый раз, рекомендую обернуть всю программу в следующую штуку:

```
#define A
#ifdef A
//code
#endif
```

Тогда, вам не придётся комментировать весь код каждый раз, а всего лишь нужно написать, например 1 у A в первой строке, это займёт меньше времени, которое так драгоценно на олимпиаде.

```
#define A1
#ifdef A
 //code
#endif
```

Аналогично можно покопаться в CLion/Codeblocks настройках и найти те же самые пункты.

2 Справка

Во-первых, общепринятая справка по C++, которая часто есть на олимпиадах — это cppreference.com или его урезанная русская версия, которая иногда написана гугл-переводчиком ru.cppreference.com.

Здесь есть вся информация о C++, нужно лишь научится ей пользоваться. Допустим, вы хотите узнать все методы std::map. Тогда ищите в поиске map и переходите по std::map.

2.1 Class template

В самом верху вас встречает шаблон класса map:

```
template<
  class Key,
  class T,
  class Compare = std::less<Key>,
  class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

И его описание:

«std::map is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees.»

Это означает, что чтобы создать map, нужно написать одно из:

```
struct Cmp {
  bool operator() (pair<int, int> first, pair<int, int> second) const {
    return first.second < second.second;
  }
}
int main() {
  map<string, int> int_by_str;
  map<int, double> double_by_int;

  map<pair<int, int>, int, Cmp> map_with_custom_comparator;
}
```

Первые два объявления стандартные и просто дают доступ по ключу типа Key значение типа T.

Третье объявление сортирует внутри map-ы элементы по ключу в соёв порядке, который вы определили в константном методе () в структуре Cmp.

Последний параметр в шаблоне — аллокатор — функция, с помощью которой C++ выделяет память. Её мы никогда использовать не будем.

2.2 Member types

Этот блок говорит, какие переменные есть в пространстве класса map<Key, T>. К ним можно обратиться следующим образом:

```
map<int, int> mp = { { 1, 1 } };

map<int, int>::iterator it = mp.begin();
```

Из важной информации — тут можно узнать, какому классу принадлежит итератор: Input/Output/Forward/Bidirectional/Random, о каждом из них более подробно можно прочитать на том же cppreference здесь.

2.3 Member functions

В первую очередь, загляните в конструктор.

Обращайте внимание на пометки *(since C++14)*. Они означают, что лишь, начиная с C++14, существует эта перегрузка, а перегрузки, помеченные *(untill C++11)*, или, что хуже, *(removed in C++17)*, как, например, в std::random_shuffle, говорят, что начиная с определённого стандарта такой функции не существует, или она навсегда удалена.

Так же рекомендую мысленно отбрасывать все аргументы типа аллокаторов, компараторов и т.п.

2.3.1 constructor

Конструктор вызывается при создании объекта, чтобы вы могли его создать несколькими способами, например:

```
map<int, int> m1;

vector<pair<int, int>> a = { { 1, 2 }, { 2, 3 } };
map<int, int> m2(a.begin() + 1, a.begin() + 2);

map<int, int> m3 = m2;

map<int, int> m4 = { { 1, 1 } };
```

Я проиллюстрировал на примерах каждый из конструкторов в том же порядке за исключением тех, что отличаются наличием компаратора и 4-го, там используется move-семантика, которая не пригодится вам в олимпиадном программировании.

Ключевое слово explicit означает, что нельзя неявно приводить к типу map, как, например, это умеет делать double:

```
double x = 1 + 0.1; // int + double -> double + double
```

Второй конструктор говорит, что он принимает два Input итератора — на begin и end списка. Напомню, что это означает полуинтервал $[begin; end)$ и Input означает, что класс вашего итератора должен быть не ниже Input.

Третий конструктор говорит, что можно создать копию уже существующего map.

Пятый — можно использовать initializer_list, это отдельная структура данных в C++ и именно она возникает, когда вы пишете что-то в фигурных скобках. Не многие знают, но перегрузка с initializer_list есть и у std::min.

Всё это можно понять по:

1. Пояснениям к каждому конструктору
2. Типам аргументов и их названиям
3. Разделу Parameters.

Ну и можно заглянуть в раздел Complexity.

2.3.2 Function template

Аналогично можно заглянуть в любую функцию, например метод insert и увидеть там ровно такую же структуру.

Стоит сказать, что есть разные ключевые слова, которые вы можете видеть перед методами.

```
struct {
  inline int get1() { return 1; }
  constexpr int get2() { return 2; }
  static int get3() { return 3; }
  int get4() const { return 4; }
  constexpr inline int get5() const { return 5; }
};
```

Ключевое слово inline ускоряет ваш код — оно подставляет текст функции прямо в место вызова и фактически не выделяется ресурсов на вызов функции. Это слово носит рекомендательный характер и компилятор может его просто проигнорировать, как с рекурсивными inline функциями.

Ключевое слово constexpr позволяет вызвать функции в моменте компиляции кода, а не только в run-time.

Ключевое слово static говорит, что это не метод функции, а отдельная функция. Просто вы захотели логически отнести эту функцию к вашему классу.

Ключевой слово const говорит, что этим методом вы не хотите менять внутренние поля в вашей структуре.

Ну и можно всё это объединять, как показывает пятый пример.

Есть ещё много ключевых слов, но это основные, которые вы можете встретить.

2.4 Что почитать в справке

- Методы у всех стандартных контейнеров: vector, set, map, muliset (обратите внимание на время работы count), unoudered_set, unoudered_map
- Стандартные алгоритмы в библиотеках algorithm и numeric
- Заранее прочитать, что есть в 20-м стандарте, чтобы быть в тренде