Фишки C++

Сапожников Денис

Содержание

1 Введение

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.

2 Классы

```
struct Vect {
  int x, y;
  Vect(int x = 0, 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*(int k) {
    return { x * k, y * k };
  }
  int 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;
  }
};
```

3 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.

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.

```
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;
}
```

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:

```
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!

4 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:

```
map cpp17 = {{"lol", 1}, {"kek", 228}, {"chebureck", -1}};
```

5 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 for each:

```
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%

6 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);
```

7 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;
```

8 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. Он перенаправляет стандартные потоки через файлы.

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>
int main() {
  for (int i = 0; i < 2; i++) {
    ifstream in((to_string(i) + ".txt").c_str());
    ofstream out(("ans" + to_string(i) + ".txt").c_str());
    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>
int main() {
  ifstream cin("input.txt");
  ofstream cout("output.txt");
  int x;
  cin >> x;
  cout << x + 1;
}
```

9 Настраиваем ваш вижак или clion

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++ → 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
```

Начиная с C++11 freopen считается небезопасным и нужно прописать ещё и дефайн _CRT_SECURE_NO_WARNINGS в свойствах проекта: правой кнопкой мыши по проекту → свойства проекта → C++ → Препроцессор → Определения препроцессора, дописываете туда оба дефайна.

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
```