Абсолютно необходимый минимум для использования LeetCode

М. А. Ложников

19 сентября 2019 г.
Ревизия: 2

Это предварительная невыверенная версия. Читайте на свой страх и риск.

Материалы этого занятия ни в коем случае не претендуют на полноту изложения материала. Цель занятия заключается в том, чтобы рассказать абсолютно необходимый минимум, который нужно знать для сдачи задач на C++ в тестирующую систему `https://leetcode.com`.

1 Оболочка над динамическим массивом. Шаблон std::vector

В языке C++ есть специальный механизм, который позволяет создавать структуры данных (классы или структуры) и функции, параметрами которых являются некоторые другие типы данных.

Например, требуется написать функцию, которая возводит число в квадрат. Для самой функции не важно, какой тип данных у её аргумента, главное, чтобы этот тип данных поддерживал операцию умножения. В C++ можно сделать сам тип данных аргумента параметром соответствующей функции.

Ещё одним примером может являться класс, который является контейнером (список, массив и т. д.) для элементов какого-либо типа данных. В таком случае сам тип данных элемента контейнера можно сделать параметром.

Такие функции или классы называют шаблонами. Разумеется, типы данных, работающие с шаблоном, должны поддерживать интерфейс, при помощи которого с ними взаимодействует шаблон.

В стандартной библиотеке C++ есть удобный шаблон `std::vector`, который является простой обёрткой над динамическим массивом. Единственным обязательным аргументом шаблона является тип данных элементов, которые хранятся в этом контейнере. Для того, чтобы использовать шаблон нужно подключить заголовочный файл `<vector>`.

Работать с ним очень просто.

```cpp
/* Объявляем вектор целых чисел. Для этого нужно написать имя шаблона (std::vector)
   и в угловых скобках перечислить его обязательные аргументы (тип данных элемента
   массива). */
std::vector<int> vectorOfIntegers;
/* Объявляем вектор чисел типа double. */
std::vector<double> vectorOfDoubles;
/* Объявляем вектор векторов целых чисел. */
std::vector<std::vector<int>> vectorOfVectorsOfIntegers;
```

Здесь важно понимать различие между шаблоном и классом. Тип `std::vector` является шаблоном, а тип `std::vector<int>` является классом. То есть, если в шаблон подставить аргументы

(инстанцирование шаблона), то получится класс. Сам шаблон классом не является. Наконец, классы `std::vector<int>` и `std::vector<double>` являются разными классами, не смотря на то, что они построены на базе одного и того же шаблона. Таким образом, эти типы не поддерживают операций сравнения между собой и нельзя из объекта первого типа создать объект второго при помощи copy-конструктора.

Отметим, что использование шаблонов не вызывает накладных расходов на этапе выполнения программы. При инстанцировании шаблона компилятор генерирует код получившегося типа данных или функции, то есть код шаблона компилируется только после инстанцирования. Следовательно, если инстанцировать шаблон разными аргументами, компилятор сгенерирует разный код. Таким образом, шаблоны могут значительно увеличить время компиляции программы.

```cpp
#include <iostream>
#include <vector>    // Это для вектора

int main() {
  /* Объявление вектора (динамического массива) int'ов. По умолчанию в векторе 0
     элементов. */
  std::vector<int> v;

  /* Метод push_back() кладёт элемент в конец вектора, то есть после последнего
     элемента. */
  for(int i = 0; i < 10; i++)
    v.push_back(i * i % 10);

  /* Метод size() возвращает кол-во эл-тов в векторе. Тип std::size_t это беззнаковый
     целочисленный тип, вмещающий в себя указатель. Он предпочтителен для измерения
     размеров массивов. */
  std::size_t vSize = v.size();
  std::cout << "Vector has " << vSize << " elements." << std::endl; // Выведет 10.

  /* Метод push_back() переодически перевыделяет память и копирует все элементы
     в новый массив большего размера. Делает он это как только свободное место в старом
     массиве заканчивается. push_back() делает ёмкость нового массива в два раза
     больше старого. Метод capacity() позволяет узнать текущую ёмкость (в эл-тах)
     выделенной памяти. Всегда выполнено size() <= capacity(). */
  std::cout << "We can insert " << v.capacity() - v.size()
            << " element(s) without reallocation" << std::endl; // Выведет 6.

  /* Вектор поддерживает чтение элемента по индексу (переопределена операция []).
     Элементы нумеруются от 0 до size() - 1. */
  std::cout << "The third element is equal to " << v[3] << std::endl;
  v[4] = 40; // И запись элемента по индексу.

  // Распечатаем вектор
  for (std::size_t i = 0; i < v.size(); i++)
    std::cout << v[i] << std::endl;

  return 0;
}
```

1.1 Изменение размера вектора

Рассмотрим пример изменения размера вектора.

```cpp
#include <iostream>
#include <vector>

/* Функция печатает вектор на экран. Функция принимает вектор по ссылке для того,
   чтобы избежать копирования аргумента. Константность нужна для того, чтобы защитить
   аргумент от случайных изменений. */
void PrintVector(const std::vector<int>& data) {
  std::cout << "Vector = [ ";
  for (std::size_t i = 0; i < data.size(); i++) {
    std::cout << data[i] << ' ';
  }
  std::cout << "]" << std::endl;
}

int main() {
  std::vector<int> v;

  for(int i = 0; i < 10; i++)
    v.push_back(i * i % 10);

  /* Метод resize() изменяет размер вектора (кол-во элементов в нём).
     Увеличим количество элементов с 10 до 15. Старые элементы сохранятся, новые станут
     равны нулю. Обычный массив НЕ ОБНУЛЯЛ ЭЛЕМЕНТЫ, а вектор обнуляет или вызывает
     конструктор по-умолчанию. Разумеется, в качестве нового размера можно поставить
     переменную (не только константу). */
  v.resize(15);

  /* Теперь увеличим размер с 15 до 20. Старые элементы сохранятся, новые элементы
     будут равны 101 (последние 5). Также произойдёт перевыделение памяти. */
  v.resize(20, 101);

  std::cout << "Enter two elements " << std::endl; // Считаем пару элементов.
  for (std::size_t i = 2; i < 4u; i++)
    std::cin >> v[i];

  // Распечатаем вектор
  for (std::size_t i = 0; i < v.size(); i++)
    std::cout << v[i] << std::endl;

  /* Можно сразу объявить вектор заданного размера.  В этом векторе будет 10 элементов,
     которые проинициализируются нулём. */
  std::vector<int> v2(10);
  PrintVector(v2);    // Выводим результат

  /* Метод assign() изменяет размер вектора и заполняет весь вектор указанными
     значениями.  Изменим размер вектора с 10 до 15 и заполним весь вектор элементами,
     равными 27. */
```

```
48   v2.assign(15, 27);
49   PrintVector(v2);     // Выводим результат
50
51   return 0;
52 }
```

1.2 Резервирование памяти в векторе

Рассмотрим следующую задачу: требуется положить в пустой вектор N элементов. Если просто объявить пустой вектор и добавлять элементы при помощи `push_back()`, то вектор будет периодически перевыделять память и копировать в новую память старые элементы. Если сразу создать вектор заданного размера, то его значения приниициализируются значениями по-умолчанию. Оба подхода приводят к накладным расходам.

Для решения задачи в векторе есть специальный метод `reserve()`, который позволяет увеличить ёмкость вектора, не меняя количество элементов в нём (то есть просто выделить память заданного размера). После этого можно закинуть элементы при помощи `push_back()`.

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5    /* Создадим пустой вектор, зарезервируем в нём память под 10 элементов. Метод
6       reserve() выделит память, то есть сделает capacity равной 10. Однако, в векторе
7       будет по-прежнему 0 элементов. Если бы в векторе были элементы до этого, то
8       reserve() бы их скопировала в новый массив. */
9    std::vector<int> v;
10
11   v.reserve(10); // В reserve() можно подставить переменную (не только константу).
12
13   // Теперь их можно закинуть туда с помощью push_back().
14   for (int i = 0; i < 10; i++)
15     v.push_back(i*i*i);
16
17   for (std::size_t i = 0; i < 10; i++)
18     std::cout << v[i] << std::endl;
19
20   return 0;
21 }
```

2 Сортировка вектора

Для сортировки вектора можно использовать функцию `std::sort()` из заголовочного файла `<algorithm>`. У этой функции два обязательных параметра `beginIt` и `endIt`, которые называются итераторами и определяют промежуток, который нужно отсортировать. Итератор это специальный класс, который написан таким образом, что ведёт себя как указатель.

Функция `std::sort()` сортирует промежуток `[beginIt, endIt)`, левая граница включена, а правая нет. У шаблона `std::vector` есть два метода `begin()` и `end()`, которые возвращают итератор

на начало вектора и итератор на конец вектора. Причём итератор на конец вектора указывает на элемент “после последнего”. Таким образом, в полуинтервале `[begin(), end())` лежат все элементы вектора.

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Это для функции std::sort

int main() {
  /* Функция std::sort библиотеки <algorithm> позволяет отсортировать промежуток
     вектора, заданный итераторами.
  */

  /* Элементы вектора можно задавать списком инициализации, начиная с C++11. */
  std::vector<int> v = {1, 4, 8, 3, 8, 2, 5, 7, 9, 5, 3, 2, 5, 8, 4, 4, 7};

  /* Сортируем первые 10 элементов вектора. Прибавление числа к итератору вектора
     работает точно так же, как и для указателей. */
  std::sort(v.begin(), v.begin() + 10);

  for (std::size_t i = 0; i < v.size(); i++)
    std::cout << v[i] << std::endl;

  std::sort(v.begin(), v.end());        // Сортируем весь вектор.

  for (std::size_t i = 0; i < v.size(); i++)
    std::cout << v[i] << std::endl;

  return 0;
}
```

3 Класс std::string для работы со строками

Тип данных `std::string`, определённый в заголовочном файле `<string>` позволяет просто работать со строками. Этот тип данных является стандартным классом для работы со строками в `C++` и входит в библиотеку `STL` (Standard template library) языка `C++`. Тип данных `std::string` имеет интуитивно понятный интерфейс. Продемонстрируем его на примере

```cpp
#include <iostream>
#include <string>
#include <cstdio> // Это для функции std::printf

int main() {
  std::string str = "Hello world!"; // Инициализация объекта из типа const char*
  // Методы length() и size() класса std::string позволяют узнать длину строки
  std::cout << "The length of string '" << str << "' is equal to "
            << str.length() << std::endl;
  /* Тип std::size_t хранит целые неотрицательные числа. Его размер позволяет хранить
```

```
    указатель. Все контейнеры C++ возвращают размер в std::size_t. */
  std::size_t size = str.size();
  std::cout << "The size is equal to " << size << std::endl;
  /* Оператор [], переопределённый для класса std::string позволяет читать и
     модифицировать отдельные символы. Нумерация ведётся с нуля. */
  str[6] = 'W';
  std::cout << "Modified string: '" << str << "'" << std::endl;

  // Строки можно считывать с помощью класса std::cin.
  std::cout << "Enter your name-> ";
  std::string name;
  std::cin >> name;
  std::cout << "Hello, " << name << "!" << std::endl;

  /* Объекты класса std::string можно сравнивать не только между собой, но и с типами
     const char* (строками в кавычках). Сравнение производится в лексикографическом
     порядке. */
  if (name == "Mikhail")
    std::cout << "Nice to meet you!" << std::endl;
  if (name < "Mikhail")
    std::cout << "Your name is inappropriate!" << std::endl;

  /* Объекты типа std::string можно складывать в том числе и со строками в кавычках
     (const char*). Результатом является объединение строк. */
  std::string result = name + " knows how to use strings in C++";
  std::cout << result << std::endl;

  /* Наконец, объекты типа std::string могут возвращать указатели на const char
    (то есть const char*), иными словами, строки в стиле языка C. Примечание:
    слово 'const' говорит о том, что элементы получившейся строки нельзя менять. */
  const char* cStyleString = result.c_str();
  std::printf("C-style: %s\n", cStyleString);

  return 0;
}
```

4 Цикл range-based for (C++11)

Для итерации по вектору часто бывает удобно использовать цикл *range-based for*. Поясним на примере.

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

int main() {
  std::vector<int> vec = { 1, 2, 4, 7, 3, 5, 6, 7, 3, 2 };

  /* Синтаксис очень простой. В круглых скобках пишется тип данных элемента вектора,
     затем имя переменной, в которую на каждой итерации будет скопирован элемент
```

```
    контейнера, затем двоеточие и имя контейнера. Нижеприведённый цикл проитерируется
    по вектору по значению от первого элемента до последнего. */
  for (int item : vec) {
    std::cout << item << std::endl;
  }

  /* Если элементы вектора долго копируются, тогда нужно итерироваться по
    константной ссылке. */
  for (const int& item : vec)
    std::cout << item << std::endl;

  /* Наконец, если требуется изменить элементы вектора, то нужно итерироваться
    по обычной ссылке. */
  for (int& item : vec)
    std::cin >> item;

  return 0;
}
```

4.1 Ключевое слово auto

В циклах *range-based for* (и не только в них) можно заставить компилятор автоматически вывести тип данных элемента. Делается это при помощи ключевого слова `auto`.

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

int main() {
  std::vector<int> vec = { 1, 2, 4, 7, 3, 5, 6, 7, 3, 2 };

  /* Итерация по значению. */
  for (auto item : vec)
    std::cout << item << std::endl;

  /* Итерация по ссылке позволяет менять элементы вектора. */
  for (auto& item : vec)
    std::cin >> item;

  return 0;
}
```

Однако, злоупотреблять этим не следует. Использование `auto` актуально в тех случаях, когда выводимый тип данных очевиден для программиста или занимает слишком много места.

5 Минимальные сведения о лямбда-выражениях (C++11)

Иногда возникает необходимость написать функцию, которая используется только в одном участке кода программы. Например, правило сравнения для сортировки. Причем, в некоторых случаях хочется, чтобы эта функция находилась перед глазами то есть, чтобы не приходилось перематывать

экран или заходить в другой файл, чтобы посмотреть как она работает. С этой целью в C++11 были введены лямбда-выражения. Лямбда-выражения — это функции, которые можно определять непосредственно в коде программы.

Синтаксис:

```
[переменные, которые будут видны в теле лямбды](аргументы лямбда-выражения) {
  // Тело лямбда-выражения.
}
```

В квадратных скобках можно указать переменные, которые должны быть доступны внутри лямбда-выражения.

5.1 Простой пример

Отсортируем вектор точек по возрастанию координаты `x`.

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

struct Point {
  int x;
  int y;
};

int main() {
  /* Внешний список инициализации для вектора, внутренние --- для его элементов. */
  std::vector<Point> points = {
    {1, 2}, {2, 5}, {5, 3}, {5, 8}, {4, 1}, {3, 7}
  };

  /* Сортируем вектор по возрастанию иксов. В качестве третьего параметра функции
     std::sort() используется лямбда-выражение, задающее правило сравнения элементов
     вектора. */
  std::sort(points.begin(), points.end(),
            [](const Point& left, const Point& right) {
    return left.x < right.x;
  });
  return 0;
}
```

Лямбда-выражение в функции `std::sort()` должно принимать два аргумента того же типа, что и элементы вектора по константной ссылке (или по значению, если они быстро копируются) и возвращать `true`, если первый аргумент меньше второго и `false` в противном случае.

5.2 Сохранение лямбда-выражения в переменную

Тот же самый пример, только правило сравнения сохраняется в переменную.

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

struct Point {
  int x;
  int y;
};

int main() {
  /* Внешний список инициализации для вектора, внутренние --- для его элементов. */
  std::vector<Point> points = {
    {1, 2}, {2, 5}, {5, 3}, {5, 8}, {4, 1}, {3, 7}
  };

  /* Сохраняем лямбду в переменную comparator. Вместо типа данных следует поставить
     auto. */
  auto comparator = [](const Point& left, const Point& right) {
    return left.x < right.x;
  };

  /* Сортируем вектор по возрастанию иксов. В качестве третьего параметра функции
     std::sort() используется лямбда-выражение, задающее правило сравнения элементов
     вектора. */
  std::sort(points.begin(), points.end(), comparator);

  return 0;
}
```

5.3 Пример сложного правила сравнения

В некоторых случаях в лямбду нужно передать какие-либо дополнительные параметры кроме тех, которые она принимает в круглых скобках. Для этих целей в квадратных скобках следует указать переменные, которые должны быть доступны в теле лямбды. Причём, доступ может быть по значению или по ссылке. При доступе по значению соответствующая переменная копируется. При доступе по ссылке лямбда-выражение может поменять соответствующую переменную.

Переменные, которые нужно дополнительно передать в лямбду указываются в квадратных скобках. Рассмотрим несколько примеров.

- `[](список аргументов)` — если квадратные скобки пусты, то кроме списка аргументов в лямбду ничего не передается.
- `[a, &b](список аргументов)` — переменная `a` передается по значению, а переменная `b` по ссылке (не путать с указателем, синтаксис такой же).
- `[&](список аргументов)` — все доступные переменные передаются по ссылке.
- `[=](список аргументов)` — все доступные переменные передаются по значению.
- `[=, &b](список аргументов)` — переменная `b` передаётся по ссылке, а все остальные доступные переменные передаются по значению.
- `[&, b](список аргументов)` — переменная `b` передаётся по значению, а все остальные доступные переменные передаются по ссылке.

В следующем примере используется более сложное правило сортировки, которое зависит от неко-

торого внешнего параметра x0.

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

struct Point {
  int x;
  int y;
};

int main() {
  /* Внешний список инициализации для вектора, внутренние --- для его элементов. */
  std::vector<Point> points = {
    {1, 2}, {2, 5}, {5, 3}, {5, 8}, {4, 1}, {3, 7}
  };

  const int x0 = 3;

  /* Немного более сложное правило сортировки. Внутри лямбда-выражения доступна
     константа x0 по значению. */
  std::sort(points.begin(), points.end(),
            [x0](const Point& left, const Point& right) {
    if (left.x <= x0 || right.x <= x0)
      return left.x < right.x;
    else
      return left.x > right.x;
  });
  return 0;
}
```

6 Список задач

Задача 1 *(Источник: https://leetcode.com/problems/partition-labels/) Дана строка, состоящая из строчных латинских символов. Требуется разбить эту строку на как можно большее количество частей таким образом, чтобы каждый символ появлялся не более чем в одной части разбиения. Требуется вывести длины подстрок получившегося разбиения.*
Пример:
Ввод: `S = "ababcbacadefegdehijhklij"`
Вывод: `[9,7,8]`
Пояснение: *Искомое разбиение состоит из* `"ababcbaca"`, `"defegde"`, `"hijhklij"`, *в нём каждая буква встречается не более чем в одной части. Другое разбиение вроде* `"ababcbacadefegde"`, `"hijhklij"` *некорректно, поскольку оно разбивает искомую строку на меньшее число частей.*

В задаче действуют следующие ***ограничения:*** *длина строки не превосходит 500 символов, все символы являются строчными латинскими.*

Задача 2 *(Источник: https://leetcode.com/problems/minimum-add-to-make-parentheses-valid/) Дана строка* `S` *из открывающихся* `(` *и закрывающихся скобок* `)`*. Требуется добавить в эту строку наименьшее*

количество открывающихся и/или закрывающихся скобок таким образом, чтобы скобочная последовательность стала корректной.

Последовательность скобок считается корректной тогда и только тогда, когда

- *Последовательность является пустой;*
- *Она может быть представлена как* `AB`*, где* `A` *и* `B` *— правильные скобочные последовательности;*
- *Она может быть представлена как* `(A)`*, где* `A` *— правильная скобочная последовательность.*

В задаче действуют следующие ***ограничения:*** *длина строки не превосходит 1000 символов, строка состоит только из символов* `(` *и* `)`*.*

Задача 3 *(Источник:* `https://leetcode.com/problems/score-after-flipping-matrix/`*) Дана $N \times M$-матрица A, содержащая значения 0 или 1. За один ход можно выбрать строку или столбец матрицы и инвертировать значения в этой строке или столбце, то есть заменить нули на единицы, а единицы на нули. После произвольного числа ходов строки матрицы интерпретируются как двоичные числа, сумма которых является количеством набранных очков. Требуется определить наибольшее количество очков, которое можно получить для заданной матрицы.*

В задаче действуют следующие ***ограничения:***

- $1 \leqslant N \leqslant 20$*;*
- $1 \leqslant M \leqslant 20$*;*
- $A_{i,j} = 0$ *или* $A_{i,j} = 1$*.*

Задача 4 *(Источник:* `https://leetcode.com/problems/largest-values-from-labels/`*). Дан набор из N элементов. Каждый элемент имеет целочисленное значение v_i и целочисленную метку l_i. Требуется выбрать подмножество элементов S таким образом, что*

- $|S| < P$*, где P — заданное число;*
- *Для каждой метки L число элементов в S с этой меткой не превосходит заданного Q.*

Выведите наибольшую возможную сумму элементов в искомом подмножестве.

В задаче действуют следующие ***ограничения:***

- $1 \leqslant N \leqslant 20000$*;*
- $0 \leqslant v_i,\ l_i \leqslant 20000$*;*
- $1 \leqslant P,\ Q \leqslant N$*.*

Задача 5 *(Источник:* `https://leetcode.com/problems/car-pooling/`*) Предположим, что есть автобус, который едет по маршруту (только в одну сторону). Изначально в автобусе имеется некоторое заданное число свободных мест (*`capacity`*). Автобус может подбирать и высаживать пассажиров. Информация о пассажирах указана в массиве* `trip`*, где* `trip[i]` *содержит три целых числа: информацию о числе пассажиров, месте, где их нужно подобрать и месте, где их нужно высадить (место указывается в километрах от начала маршрута). Автобус не может разворачиваться и не может посадить больше людей, чем количество свободных мест.*

Требуется определить, сможет ли автобус подобрать всех людей и отвезти их куда надо.

В задаче действуют следующие ***ограничения:***

- `trips.size() <= 1000`*;*
- `trips[i].size() == 3`*;*

- `1 <= trips[i][0] <= 100`;
- `0 <= trips[i][1] < trips[i][2] <= 1000`;
- `1 <= capacity <= 100000`.

Задача 6 *(Источник: https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/)* *Дан целочисленный массив* `prices` *длины* `N`, *в* `i`*-ом элементе которого записана стоимость товара в* `i`*-ый день, а также неотрицательное целое* `fee`, *представляющее собой налоговые сборы при продаже. Можно делать сколько угодно транзакций, однако нельзя покупать более одной единицы товара (перед тем как купить нужно сначала продать), и с каждой продажи необходимо дополнительно оплатить налоговый сбор* `fee`. *Подсчитайте наибольшую прибыль, которую можно получить.*

В задаче действуют следующие ***ограничения:***

- `0 < N <= 50000`;
- `0 < prices[i] < 50000`;
- `0 <= fee < 50000`.

Задача 7 *(Источник: https://leetcode.com/problems/boats-to-save-people/) Дан массив* `people` *длины* N, *в* i*-ом элементе которого содержится вес* i*-ого человека. Каждая лодка способна выдержать вес не более* `limit`. *Выведите наименьшее количество лодок, в которых можно разместить всех человек.*

В задаче действуют следующие ***ограничения:***

- `1 <= N <= 50000`;
- `1 <= people[i] <= limit <= 30000`.

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`

[3] `https://en.cppreference.com/w/cpp/container/vector`

[4] `https://en.cppreference.com/w/cpp/string/basic_string`