Векторы и строки

Вектор (`std::vector`) и строка (`std::string`) — это важные базовые контейнеры стандартной библиотеки C++. Они хранят свои элементы в непрерывном фрагменте памяти. Оба этих контейнера предоставляют доступ к элементам по индексу и позволяют эффективно добавлять новые элементы в конец.

Контейнер `std::vector`

В стандартной библиотеке C++ *вектором* (std::vector) называется динамический массив, обеспечивающий быстрое добавление новых элементов в конец и меняющий свой размер при необходимости. Вектор гарантирует отсутствие утечек памяти (об этом мы поговорим в других параграфах, сейчас просто считайте, что это хорошо).

Для работы с вектором нужно подключить заголовочный файл `vector`.

Элементы вектора должны быть одинакового типа, и этот тип должен быть известен при компиляции программы. Он задаётся в угловых скобках после `std::vector`: например, `std::vector<int>` — это вектор целых чисел типа `int`, а `std::vector<std::string>` — вектор строк.

Само имя `std::vector` не является типом данных: это *шаблон*, в который требуется подставить нужные параметры (тип элемента), чтобы получился конкретный тип данных. Подробнее о том, что такое шаблоны и как их применять, мы расскажем в параграфе «Шаблоны».

Рассмотрим пример программы, которая заполняет вектор элементами и печатает их через пробел:

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

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    for (int elem : data) {
```

```
        std::cout << elem << " ";
    }
    std::cout << "\n";
}
```

Здесь мы инициализируем вектор через список инициализации, в котором элементы перечислены через запятую. Другой способ инициализации вектора — указать число элементов и (при необходимости) образец элемента:

```
#include <string>
#include <vector>

int main() {
    std::vector<std::string> v1;  // пустой вектор строк
      std::vector<std::string> v2(5);   // вектор из пяти пустых строк
     std::vector<std::string> v3(5, "hello");  // вектор из пяти строк "hello"
}
```

Обращение к элементам

Выше мы использовали для печати элементов вектора цикл `range-for`. Но иногда удобнее работать с индексами. Вектор хранит элементы в памяти последовательно, поэтому по индексу элемента можно быстро найти его положение в памяти. Индексация начинается с нуля:

```
std::vector<int> data = {1, 2, 3, 4, 5};
int a = data[0];  // начальный элемент вектора
int b = data[4];  // последний элемент вектора (в нём пять элементов)
data[2] = -3;  // меняем элемент 3 на -3
```

Чтобы узнать общее количество элементов в векторе, можно воспользоваться функцией `size`:

```
std::cout << data.size() << "\n";
```

Отрицательные индексы, как в некоторых других языках программирования, не допускаются.

Обратите внимание: когда мы обращаемся по индексу через квадратные скобки, проверки его корректности не происходит.Это ещё одно проявление принципа *«мы не должны платить за то, что не используем»*.

Встроенные валидаторы замедляют программу: предполагается, что программист пишет правильный код и уверен, что индекс `i` в выражении `data[i]` неотрицателен и удовлетворяет условию `i < data.size()`. В этом случае они ему не нужны.

Если всё же обратиться к вектору по некорректному индексу, то программа во время выполнения попадёт в неопределённое поведение: фактически она попробует прочитать память, не принадлежащую вектору.

Если вам не хочется делать много лишних проверок, а в корректности индекса вы не уверены, то можно использовать функцию `at`:

```
std::vector<int> data = {1, 2, 3, 4, 5};
std::cout << data[42] << "\n";     // неопределённое поведение: может произойти всё что угодно
std::cout << data.at(0) << "\n";  // напечатается 1
std::cout << data.at(42) << "\n";  // произойдёт исключение std::out_of_range — его можно будет перехватить и обработать
```

Про работу с исключениями мы поговорим отдельно в параграфе «Обработка исключений».

Рассмотрим функции вектора `front` и `back`, которые возвращают его первый и последний элемент без использования индексов:

```
std::vector<int> data = {1, 2, 3, 4, 5};
std::cout << data.front() << "\n";  // то же, что data[0]
std::cout << data.back() << "\n";     // то же, что
```

```
data[data.size() - 1]
```

Важно учитывать, что вызов этих функций на пустом векторе приведёт к неопределённому поведению.

Для проверки вектора на пустоту вместо сравнения `data.size() == 0` принято использовать функцию `empty`, которая возвращает логическое значение:

```
if (!data.empty()) {
    // вектор не пуст, с ним можно работать
}
```

Итерация по индексам

Так сложилось, что в стандартной библиотеке индексы и размеры контейнеров имеют беззнаковый тип. Вместо `unsigned int` или `unsigned long int` для него используется традиционный псевдоним `size_t` (а точнее, `std::vector<T>::size_type`). Тип `size_t` на самом деле совпадает с `uint32_t` или `uint64_t` в зависимости от битности платформы. Его использование в программе дополнительно подчёркивает, что мы имеем дело с индексами или с размером.

Итерацию по элементам `data` с помощью индексов можно записать так:

```
for (size_t i = 0; i != data.size(); ++i) {
    std::cout << data[i] << " ";
}
```

Это каноническая форма записи такого цикла: в ней принято использовать сравнение `!=` и префиксный `++i`. Для целых чисел не будет разницы, если написать это как-то иначе (например, через `<` и постфиксный `i++`), но потом, когда вы будете писать аналогичные циклы для итераторов других контейнеров, разница появится. Давайте привыкнем всегда оформлять цикл по индексам так.

Беззнаковость типа возвращаемого значения функции `size` порождает следующую проблему. По правилам, унаследованным ещё от языка C, результат арифметических действий над беззнаковым и знаковым типами приводится к беззнаковому типу. Поэтому выражение `data.size() - 1`, например, тоже будет беззнаковым. Если `data.size()` окажется нулём, то такое выражение будет вовсе не минус единицей, а самым большим беззнаковым целым (для 64-битной платформы это $2^{64}-1$).

Рассмотрим следующий ошибочный код, который проверяет, есть ли в векторе дубликаты, идущие подряд:

```
// итерация по всем элементам, кроме последнего:
for (size_t i = 0; i < data.size() - 1; ++i) {
    if (data[i] == data[i + 1]) {
            std::cout << "Duplicate value: " << data[i] <<
"\n";
    }
}
```

Эта программа будет некорректно работать на пустом векторе. Условие `i < data.size() - 1` на первой итерации окажется истинным, и произойдёт обращение к элементам пустого вектора. Правильнее было бы переписать это условие через `i + 1 < data.size()` или воспользоваться внешней функцией `std::ssize`, которая появилась в C++20. Она возвращает знаковый размер вектора:

```
for (std::int64_t i = 0; i < std::ssize(data) - 1; ++i) {
    if (data[i] == data[i + 1]) {
            std::cout << "Duplicate value: " << data[i] <<
"\n";
    }
}
```

Добавление и удаление элементов

В вектор можно эффективно добавлять элементы в конец и удалять их с конца. Для этого существуют функции `push_back` и `pop_back`. Рассмотрим программу, считывающую числа с клавиатуры в вектор и затем удаляющую все нули в конце:

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

int main() {
    int x;
    std::vector<int> data;
      while (std::cin >> x) {   // читаем числа, пока не закончится ввод
         data.push_back(x);  // добавляем очередное число в вектор
    }

    while (!data.empty() && data.back() == 0) {
          // Пока вектор не пуст и последний элемент равен нулю
        data.pop_back();  // удаляем этот нулевой элемент
    }
}
```

Добавление элементов в другие части вектора или их удаление неэффективно, так как требует сдвига соседних элементов. Поэтому отдельных функций, например, для добавления или удаления элементов из начала у вектора нет. Это можно сделать с помощью общих функций `insert`/`erase` и итераторов. Мы рассмотрим такие примеры позже.

Удалить все элементы из вектора можно с помощью функции `clear`.

Резерв памяти

Вектор хранит элементы в памяти в виде непрерывной последовательности, друг за другом. При этом в конце последовательности резервируется дополнительное место для быстрого добавления новых элементов. Когда этот резерв заканчивается, при вставке очередного элемента происходит *реаллокация*: элементы вектора копируются в новый, более просторный блок памяти.

Реаллокация — довольно дорогая процедура, но если она происходит достаточно редко, то её влияние незначительно. Можно доказать, что если размер нового блока выбирать в два раза больше предыдущего размера, то амортизационная сложность добавления элемента будет константной.

Текущий резерв вектора можно узнать с помощью функции `capacity` (не путайте её с функцией `size`).

Рассмотрим программу, в которой в вектор последовательно добавляются элементы и после каждого шага печатается размер и резерв:

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

int main() {
    std::vector<int> data = {1, 2};
     std::cout << data.size() << "\t" << data.capacity() <<
"\n";

    data.push_back(3);
     std::cout << data.size() << "\t" << data.capacity() <<
"\n";

    data.push_back(4);
     std::cout << data.size() << "\t" << data.capacity() <<
"\n";
```

```
    data.push_back(5);
     std::cout << data.size() << "\t" << data.capacity() << "\n";
}
```

Вот вывод этой программы:

```
2       2
3       4
4       4
5       8
```

Видно, что размер вектора увеличивается на единицу, а резерв удваивается после исчерпания. Так, при добавлении четвёрки используется имеющаяся в резерве память, а при добавлении тройки и пятёрки происходит реаллокация.

Иногда требуется заполнить вектор элементами, причём число элементов известно заранее. В таком случае можно сразу зарезервировать нужный размер памяти с помощью функции `reserve`, чтобы при добавлении элементов не происходили реаллокации. Пусть, например, нам сначала задаётся число слов, а потом сами эти слова, и нам требуется сложить их в вектор:

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

int main() {
    std::vector<std::string> words;

    size_t words_count;
    std::cin >> words_count;

      // Размер вектора остаётся нулевым, меняется только резерв:
    words.reserve(words_count);
```

```
    for (size_t i = 0; i != words_count; ++i) {
        std::string word;
        std::cin >> word;
        // Все добавления будут дешёвыми, без реаллокаций:
        words.push_back(word);
    }
}
```

Если передать в `reserve` величину меньше текущего резерва, то ничего не поменяется — резерв останется прежним.

Функцию `reserve` не следует путать с функцией `resize`, которая меняет количество элементов в векторе. Если аргумент функции `resize` меньше текущего размера, то лишние элементы в конце вектора удаляются. Если же он больше текущего размера, то при необходимости происходит реаллокация и в вектор добавляются новые элементы с дефолтным значением данного типа.

```
#include <vector>

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

       data.reserve(10);    // поменяли резерв, но размер вектора остался равным пяти

    data.resize(3);  // удалили последние элементы 4 и 5
    data.resize(6);  // получили вектор 1, 2, 3, 0, 0, 0
}
```

Многомерные векторы

Воспользуемся вектором векторов, чтобы сохранить матрицу (таблицу) целых чисел. Пусть на вход программы сначала поступают число строк и число столбцов матрицы, а потом — сами элементы построчно:

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

int main() {
    size_t m, n;
    std::cin >> m >> n;  // число строк и столбцов

    // создаём матрицу matrix из m строк, каждая из которых
— вектор из n нулей
                std::vector<std::vector<int>>    matrix(m,
std::vector<int>(n));

    for (size_t i = 0; i != m; ++i) {
        for (size_t j = 0; j != n; ++j) {
            std::cin >> matrix[i][j];
        }
    }

    // напечатаем матрицу, выводя элементы через табуляцию
    for (size_t i = 0; i != m; ++i) {
        for (size_t j = 0; j != n; ++j) {
            std::cout << matrix[i][j] << "\t";
        }
        std::cout << "\n";
    }
}
```

В этом примере мы заранее создали матрицу из нулей, а потом просто меняли её элементы.

Сортировка вектора

Рассмотрим типичную задачу — отсортировать вектор по возрастанию. Для этого в стандартной библиотеке в заголовочном файле `algorithm` есть готовая функция `sort`. Гарантируется, что сложность её работы в худшем случае составляет $O(nlogn)O(n\log n)$,

где nn — число элементов в векторе. Типичные реализации используют алгоритм сортировки Introsort.

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

int main() {
    std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6};

    // Сортировка диапазона вектора от начала до конца
    std::sort(data.begin(), data.end());

    // получим вектор 1, 1, 2, 3, 4, 5, 6, 9
}
```

В функцию `sort` передаются так называемые *итераторы*, ограничивающие рассматриваемый диапазон. В нашем случае мы передаём диапазон, совпадающий со всем вектором, от начала до конца. Соответствующие итераторы возвращают функции `begin` и `end` (не путать с `front` и `back`!). Итераторы можно считать обобщёнными индексами (но они могут быть и у контейнеров, не допускающих обычную индексацию). Подробнее про итераторы мы поговорим в отдельном параграфе.

Для сортировки по убыванию можно передать на вход *обратные итераторы* `rbegin()` и `rend()`, представляющие элементы вектора в перевёрнутом порядке:

```
std::sort(data.rbegin(), data.rend());  // 9, 6, 5, 4, 3, 2, 1, 1
```

В C++20 доступен более элегантный способ сортировки через `std::ranges::sort`:

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

int main() {
    std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6};
```

```
    std::ranges::sort(data);  // можно передать сам вектор, а не его диапазоны
}
```

Для сортировки по умолчанию используется сравнение элементов с помощью оператора <. Этот оператор работает и для самих векторов: они сравниваются лексикографически. Поэтому можно без проблем отсортировать, например, строки в матрице (векторе векторов целых чисел).

Строки

Контейнер `std::string` можно рассматривать как особый случай вектора символов `std::vector<char>`, имеющий набор дополнительных функций. В частности, у строки есть все те же рассмотренные нами функции, что и у вектора (например, `pop_back` или `resize`). Рассмотрим некоторые специфические функции строки:

```
#include <iostream>
#include <string>

int main() {
    std::string s = "Some string";

    // приписывание символов и строк
     s += ' ';  // добавляем отдельный символ в конец, это аналог push_back
    s += "functions";  // добавляем строку в конец
    std::cout << s << "\n";  // Some string functions

    // выделение подстроки
      // подстрока "string" из 6 символов начиная с 5-й позиции
    std::string sub1 = s.substr(5, 6);
    // подстрока "functions" с 12-й позиции и до конца
    std::string sub2 = s.substr(12);

    // поиск символа или подстроки
```

```
    size_t pos1 = s.find(' ');  // позиция первого пробела, в данном случае 4
    size_t pos2 = s.find(' ', pos1 + 1);  // позиция следующего пробела (11)
    size_t pos3 = s.find("str");  // вернётся 5
    size_t pos4 = s.find("#");  // вернётся std::string::npos
}
```

Вставку, замену и удаление подстрок можно сделать через указание индекса начала и длины подстроки:

```
#include <iostream>
#include <string>

int main() {
    std::string s = "Some string functions";

    // вставка подстроки
    s.insert(5, "std::");
    std::cout << s << "\n";  // Some std::string functions

    // замена указанного диапазона на новую подстроку
    s.replace(0, 4, "Special");
    std::cout << s << "\n";  // Special std::string functions

    // удаление подстроки
    s.erase(8, 5);  // Special string functions
}
```

Аналогичные действия для других контейнеров (например, для того же вектора) можно сделать через итераторы. Мы рассмотрим такие примеры в одном из следующих параграфов.

В C++20 появились удобные функции `starts_with` и `ends_with` для проверки префикса или суффикса строк:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string phrase;
    std::getline(std::cin, phrase);

    if (phrase.starts_with("hello")) {
        std::cout << "Greeting\n";
    }

    if (phrase.ends_with("bye")) {
        std::cout << "Farewell\n";
    }
}
```