Эффективные линейные контейнеры — конспект темы

Эффективный вектор

Главное свойство вектора — это последовательное расположение всех элементов в памяти.

```
vector<int> v = {1, 2, 3};
v.push_back(4);
```

Объект `v` содержит информацию о векторе:

- `int* data` — указатель на выделенный на куче отрезок памяти для трёх элементов типа `int`;
- `size` — количество элементов, которые в данный момент находятся в векторе;
- `capacity` — количество элементов, для которых потенциально есть место в выделенной памяти, тоже три.

Данные всех трёх параметров можно получить через одноимённые методы вектора.

Для освобождения от мусора есть специальный метод. Он нечасто используется и называется

Метод `shrink_to_fit` освобождает память от мусора: переаллоцирует память так, чтобы `capacity` стала равна `size`. Этот запрос необязательно будет выполнен.

Если заранее знаете, сколько элементов должно быть в векторе, используйте для добавления элементов метод `reserve`.

Эффективный дек

Дек — двунаправленная очередь. В отличие от вектора, дек не обещает хранить элементы в памяти подряд.

Создадим дек, добавим в него три элемента при инициализации, потом ещё один в конец и один в начало:

```
deque<int> d = {1, 2, 3};
d.push_back(4);
d.push_front(0);
```

В стеке создан объект `d`, где хранится количество элементов, расположенных в памяти. Внутри дека есть вектор, хранящий указатели на участки памяти, где находятся элементы. Иначе эти участки называются чанками.

При вызове `push_back` дек выделит новый чанк. Положим туда новый элемент на первое место.

При вызове `push_front` дек в очередной раз выделит новый чанк и положит туда 0, но не в начало, а в конец. Первые два элемента этого чанка пока останутся пустыми.

Внутри дек держит специальный параметр "shift". Этот параметр говорит, насколько первый элемент дека отстоит от начала чанка.

Инвалидация указателей и ссылок

- При вставке/удалении в вектор указатели и ссылки инвалидируются всегда.
- При вставке/удалении в начало или в конец дека указатели и ссылки сохраняются.
- При вставке/удалении из середины дека указатели и ссылки инвалидируются.

Инвалидация итераторов

Операции с изменением числа элементов инвалидируют итераторы и у вектора, и у дека.

Эффективный список

В отличие от односвязного списка, по двусвязному списку можно перемещаться не только вперёд, но и назад.

В векторе все элементы расположены последовательно в одном участке памяти. В деке элементы находятся в памяти в небольших чанках.

Удаление из дека или вектора линейное, а из списка — константное. Вставка и удаление из середины работают для списка быстрее, чем для вектора и дека.

Быстро получить доступ к элементу списка по индексу невозможно. У списка двунаправленный итератор, а не итератор произвольного доступа. Поэтому функцию бинарного поиска для списка использовать нельзя.

Алгоритм `reverse` существует в виде метода списка, и в реализации этого метода нет переставления элементов местами. Метод `reverse` просто переставляет указатели на соседние элементы.

Устройство списка позволяет оставлять итераторы рабочими. Ни вставка, ни удаление, ни какой-либо другое изменение не инвалидируют итераторы списка — если вы не удалили тот элемент, на который итератор указывал.

Ещё менее популярный контейнер, основанный на том же принципе, —

`forward_list` — однонаправленный список. Он подходит, если нужно итерировать по элементам только в одну сторону, так как в этом случае экономится память. Каждый элемент помнит указатель только на следующий элемент, но не на предыдущий.

Проще и быстрее: std::array

Желательно выделять память «по запросу», а не резервировать лишнюю.

Контейнер `std::array` — массив, который не выделяет память в куче, а хранит всё на стеке функции:

```
#include <array>
#include <iostream>

using namespace std;

int main() {
    // создадим на стеке переменные x и y,
    // положим между ними массив, заполненный восьмёрками
    int x = 111111;
    array<int, 10> numbers;
    numbers.fill(8);
    int y = 222222;
    // пройдёмся по адресам между y и x
    // и выведем то, что лежит в памяти
    for (int* p = &y; p <= &x; ++p) {
        cout << *p << " "s;
    }
    cout << endl;
```

```
    return 0;
}
```

Результат работы программы:

```
222222 8 8 8 8 8 8 8 8 8 6 0 111111
```

Массив работает в разы быстрее вектора.

Помощник в работе со строками — std::string_view

`string_view` — указатель на начало некой строки и её длина. Это не контейнер в прямом смысле: он не содержит элементы, а просто указывает на некую последовательность символов в памяти.

```
// изменим название функции
// предыдущая её версия нам ещё пригодится

// пусть теперь наша функция возвращает вектор элементов string_view
vector<string_view> SplitIntoWordsView(const string& str) {
    vector<string_view> result;
    // 1
    int64_t pos = 0;
    // 2
    const int64_t pos_end = str.npos;
    // 3
    while (true) {
        // 4
        int64_t space = str.find(' ', pos);
        // 5
        result.push_back(space == pos_end ? str.substr(pos) : str.substr(pos, space - pos));
        // 6
        if (space == pos_end) {
            break;
        } else {
            pos = space + 1;
        }
    }

    return result;
}
```

Реализация по пунктам:

1. Создаём переменную, где будем сохранять начальную позицию для поиска следующего пробела.
2. Чтобы остановить поиск, нужен аналог итератора на конец. У `string_view` такую роль выполняет `npos` — специальная константа класса. Внутри это просто большое число, которое вряд ли когда-нибудь сможет оказаться реальной позицией в строке.
3. Используем бесконечный цикл. Выходим из него, применяя `break`, когда дойдём до конца строки.
4. В цикле ищем следующий пробел, вызывая метод `find`, который вернёт расстояние до ближайшего пробела, или `npos`, если пробел не найден.
5. Метод `substr` для `string_view` возвращает `string_view`, то есть новая строка не создаётся. Продолжаем с указателями на уже имеющуюся строку. Если пробел не найден, добавляем в вектор всё, начиная с `pos`. Если найден, выделяем слово от `pos` длиной `space - pos`.
6. Достигнув конца строки, выходим из цикла. Иначе, пропускаем одну позицию — пробел — и начинаем следующую итерацию цикла.

Компилятор умеет превращать строки в объекты типа `string_view`, но сделать обратное превращение он не может. В функцию, ожидающую `string_view`, можно передать и строку. Все преобразования будут выполнены автоматически. Если функция ожидает аргумент-строку, передать туда `string_view` уже не удастся.