

Программирование на C++. Часть 2

Лекция 9

ПМИ Семестр 2

Демяненко Я.М.

2026

Обобщённый подход

- **Обобщённое программирование (generic programming)** заключается в описании структур данных и алгоритмов в терминах типов, подставляемых в качестве параметров в эти алгоритмы и структуры
- Обобщённое программирование является одним из проявлений полиморфизма
- Обобщённый подход в языке C++ основывается на шаблонах функций и шаблонах классов

Стандартная библиотека шаблонов

Внедрение механизма шаблонов в реализацию стандартной библиотеки C++ привело к появлению STL (Standard Template Library) — стандартной библиотеки шаблонов

Архитектура STL была разработана **Александром Степановым**, Менг Ли и другими сотрудниками AT&T Bell Laboratories и Hewlett-Packard Research Laboratories в начале 90-х годов

С 1998 года библиотека STL вошла в стандарты C++

Стандарт языка не называет её STL, так как эта библиотека стала неотъемлемой частью языка

Однако до сих пор часто используется это название, чтобы отличать её от остальной части стандартной библиотеки

В 1995 году получил Dr.Dobb's Excellence In Programming Award за создание STL, поделив премию с Линусом Торвальдсом.



16 ноября 1950 (76 лет)

Шесть основных видов компонентов STL

- контейнеры — объекты, которые хранят коллекции других объектов
- итераторы — объекты доступа к содержимому контейнеров; они исполняют роль посредников между контейнерами и алгоритмами
- алгоритмы — позволяют организовать обработку объектов вне зависимости от вида их организации в контейнерах
- адаптеры — используются для изменения интерфейсов других компонентов STL
- функторы (функциональный объекты) — объекты, каждый из которых скрывает функцию с целью передачи её в качестве параметра в алгоритмы
- аллокаторы — используются контейнерами для выделения памяти объектам контейнеров и её освобождения

Главное об обобщённых алгоритмах STL

Поскольку они **могут использоваться со многими или даже со всеми контейнерами**, отпадает необходимость в определении соответствующих функций-членов у отдельных контейнеров, что **снижает размер кода** и **упрощает интерфейсы** контейнеров

Тем не менее, если **эффективность** алгоритма **сильно зависит от внутренней реализации**, он обычно реализуется как **функция-член шаблона класса**

Нет наследованию в шаблонах

Разработчики отказались от использования наследования при реализации библиотеки STL

В частности, деструкторы в шаблонах классов не являются виртуальными

В связи с этим не рекомендуется разрабатывать собственные шаблоны контейнеров через наследование шаблонов STL

Общая характеристика STL

Вся библиотека состоит **только из заголовочных** файлов

В контейнерах можно использовать **любые типы** данных

Поскольку алгоритмы работают через итераторы, они могут быть применены к любым контейнерам, допускающим использование соответствующих итераторов. Этим обеспечивается **универсальность**

Однако шаблоны приводят к **большему времени компиляции**

Хотя **исполняемый код** может получиться **более эффективным**

Использование библиотеки STL **увеличивает скорость написания программ**, поскольку предоставляет готовые решения по организации данных и реализации алгоритмов

Это позволяет создавать **компактный код** программы

С другой стороны это влечёт **сложность отладки из-за тяжёлых сообщений об ошибках**

Если коротко

1. Компактность
2. Эффективность в ущерб безопасности
3. Не используются наследование и полиморфизм для эффективности
4. Универсальность
5. 0 байт в откомпилированном виде т.к. вся библиотека состоит из заголовочных файлов
6. Тяжелые сообщения об ошибках

Контейнеры

Контейнер — это набор однотипных элементов, реализующий абстрактный тип данных. Простейшим прототипом контейнера в классическом языке C++ является массив

STL вводит целый ряд разнообразных типов контейнеров:

- последовательные
- ассоциативные
- контейнеры-адаптеры
- псевдоконтейнеры

Последовательные контейнеры

- vector
- array
- list
- forward_list
- deque

Ассоциативные контейнеры

- map
- multimap
- unordered_map
- unordered_multimap
- set
- multiset
- unordered_set
- unordered_multiset

Контейнеры-адаптеры

ограничивают интерфейс базовых контейнеров —

- `queue`
- `priority_queue`
- `stack`

Псевдоконтейнеры

это шаблоны классов, похожие на стандартные контейнеры,
но удовлетворяющие не всем требованиям для стандартных контейнеров.

К ним относятся

- `bitset`
- `basic_string`
- `valarray`
- `vector<bool>`

Как связаны контейнеры и алгоритмы? Что они знают друг про друга

Как связаны контейнеры и алгоритмы? Что они знают друг про друга

Мощь STL заключается в том, что **алгоритмы** ничего не знают про **контейнеры**, а контейнеры ничего не знают про алгоритмы.

Для того, чтобы склеить контейнеры и алгоритмы в STL существуют **итераторы**.

Итератор это

Итератор это

любой тип, к которому применимы операции:

- продвижение: ++
- обращение к элементу, на который указывает итератор: *
- проверка равенства (==) и неравенства (!=)

Требования к типам элементов контейнера

Как правило, элементы, хранимые в контейнерах STL, могут быть практически **любого типа**, если их **можно копировать**, т.е. **имеют семантику значения**.

Если необходимо использовать собственный тип, то для него должны быть обязательно определены:

- конструктор без параметров,
- конструктор копии,
- деструктор,
- операция присваивания.

Дополнительно в некоторых случаях требуется определить `operator==` и `operator<`. Например, для каких-то типов контейнеров или для выполнения некоторых операций с контейнерами или их элементами.

Требования к контейнерам, которые можно использовать с обобщёнными алгоритмами библиотеки

Должны иметь:

- конструктор, создающий пустой контейнер
- конструктор копирования
- деструктор, который освобождает всю память, использованную контейнером, и вызывает деструктор каждого элемента в контейнере (деструктор контейнера не виртуален)

Возможно

- конструктор с параметрами – итераторами, задающими диапазон элементов в другом контейнере
- частные виды конструкторов в зависимости от типа контейнера

Общий вид конструктора для контейнера с элементами типа T

```
// создание пустого контейнера  
container<T> c;
```

```
// создание копии контейнера  
container<T> c2(c);
```

```
// создание контейнера с инициализацией элементами из диапазона [b,e) другого контейнера  
// диапазон задается через итераторы  
container<T> c3(b,e);
```

Чтобы использовать контейнер, нужно

- Подключение заголовочного файла (контейнер — заголовочный файл)

```
#include <vector>
```

```
#include <list>
```

```
#include <set>
```

- Подключение пространства имён
`using namespace std;`

Инициализация контейнеров

```
//пустые контейнеры
```

```
list<int> il;
```

```
set<char> cs;
```

```
//контейнеры с инициализацией списком значений
```

```
vector<int> x{2, 5, 1, 3};
```

```
list<int> y{2, 5, 1, 3};
```

```
set<int> z{2, 5, 1, 3};
```

```
//контейнеры, созданные конструктором копии
```

```
vector<int> x_copy(x);
```

```
list<int> y_copy(y);
```

```
set<int> z_copy(z);
```

```
// частный случай конструктора для вектора
```

```
// задается количество элементов и значение для всех элементов
```

```
vector<int> v2(10,0);
```

```
// Инициализация диапазоном другого контейнера
```

```
int a[] = {3,5,4,2,7};
```

```
vector<int> v(a,a+5);
```

```
vector<int> v1(a, a + 3);
```

```
list<double> l(a+2,a+5);
```

```
list<double> l2(begin(l), begin(l)++);
```

```
list<double> l3 (end(a) - 3, end(a));
```

Операции над контейнерами

Все контейнеры должны поддерживать операции присваивания и сравнения (=, ==, <, <=, !=, >, >=)

Операции сравнения применимы только к контейнерам одного типа и выполняются в лексикографическом порядке

if (v1 < v) ...; // лексикографическое сравнение

Присваивание возможно только для объектов одного типа

v = v1; // v, v1 – vector

l = l1; // l, l1 – list

Для тех случаев, когда **operator= не подходит**, используется функция **assign**, которая позволяет заполнить контейнер новыми данными. Например,

v.assign(begin(l), end(l)); //разные типы контейнеров: v – vector, l – list

l.assign(a, a+3); //перезаполнение существующего списка частью другого контейнера

l.assign(end(a) - 3, end(a));

Чтобы контейнеры можно было обрабатывать алгоритмами STL, необходимо наличие функций, возвращающих итераторы

- **iterator begin()** — возвращает итератор, указывающий на первый элемент контейнера
- **const_iterator begin() const** — возвращает константный итератор, указывающий на первый элемент контейнера
- **iterator end()** — возвращает итератор, указывающий на позицию, следующую за последним элементом контейнера
- **const_iterator end() const** — возвращает константный итератор, указывающий на позицию, следующую за последним элементом контейнера

Все контейнеры должны предоставлять ещё ряд функций

bool empty() const — возвращает true, если контейнер пуст

void clear() — очищает контейнер, делая его размер = 0

size_type max_size() const — возвращает максимальное количество элементов, которое может содержать контейнер

size_type size() const — возвращает количество элементов, в текущий момент хранящихся в контейнере

void swap(ContainerType c) — обменивает между собой содержимое двух контейнеров одного типа

Какой контейнер выбрать?

Эффективность операций для различных контейнеров **разная** и зависит от **внутреннего представления** контейнера

Поэтому для каждой конкретной задачи нужно выбирать наиболее подходящий контейнер, исходя из того, какие именно операции при решении задачи будут использоваться чаще всего

Последовательные контейнеры

Сохраняется тот порядок, в котором элементы добавляются в контейнер, т.е. в них организуется хранение элементов в линейном порядке

<code>vector</code>	является реализацией динамического массива
<code>array</code>	та же семантика, что и у массивов фиксированного размера
<code>list</code>	двунаправленный список
<code>forward_list</code>	однонаправленный список
<code>deque</code>	двусторонняя очередь

vector

Использование

в задачах, требующих **доступ** к элементу **по индексу**

Стоимость операций

- операции **вставки и удаления в конце** вектора выполняются за **постоянное** время
- **вставка и удаление внутри** вектора имеют **линейную** сложность

array

начиная с версии C++11, с целью повышения эффективности в случае работы со статическим массивом

нетиповой параметр шаблона, позволяющий выделить память на этапе компиляции

```
array<int, 4> Myarray;  
array<int, 4> Myarray1 = {5,6,7,8};
```

размер и эффективность `array<T,N>` такие же, как у статического массива `T[N]`

в отличие от остальных контейнеров, `array` не позволяет управлять размещением элементов через аллокаторы, и для него не определены операции, изменяющие размер массива

предоставляет некоторые возможности стандартных контейнеров, такие как:

- знание собственного размера,
- поддержка присваивания,
- итераторы произвольного доступа и т.д.

Итераторы для vector и array

Контейнеры vector и array поддерживают итераторы **произвольного доступа**:

- как **прямой**,
- так и **обратный**

deque

позволяет **быстро** выполнять операции **вставки/удаления** элементов **в начале и в конце** контейнера

в отличие от `vector` и `array` контейнер `deque` обычно реализован с помощью **двустороннего списка массивов фиксированного размера**

расширение `deque` дешевле, чем расширение `vector`, потому что оно не требует копирования существующих элементов в новый участок памяти

Стоимость операций

вставки/удаления в начале и конце двусторонней очереди — **постоянная**
произвольного доступа — **постоянная**
вставки/удаления внутри контейнера — **линейная**

Доступ к элементам контейнеров `vector`, `array` и `deque` по индексу

реализован двумя способами:

- перегруженной операцией `operator[](i)`
- функцией `at(i)`

Обе реализации подобны.

Отличие заключается лишь в том, что функция `at(i)` контролирует выход за границу массива, выбрасывая в случае ошибки исключение `out_of_range`.

Вектор — резюме

- Вектор — это контейнер, который является последовательным, т.е. организует **хранение** элементов в **линейном** порядке
- Вектор поддерживает **итераторы произвольного** доступа
- Операции вставки и удаления в конце вектора выполняются за постоянное время
- Операции вставки и удаления внутри вектора имеют линейную сложность
- Управление памятью выполняется автоматически

Пример. Проверить вектор целых чисел на симметричность относительно середины

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

using namespace std;

int main() {
    vector<int> v;
    int n,a;
    cin >> n;
    for (int i = 0; i < n; ++i) { //заполнение
        cin >> a;
        v.push_back(a);
    }
    ...
}
```

`v.push_back(a)` — добавление в конец

имеет постоянную сложность

если вектор заполнен (размер равен ёмкости), выполнение функции `push_back` автоматически приводит к увеличению ёмкости

если заранее известно количество вводимых элементов, то вместо `v.push_back(a)` можно использовать операцию обращения по индексу к элементу массива

```
vector<int> v(10);  
for (i=0; i<v.size(); ++i)  
    cin>>v[ i ];
```

Продолжение решения

```
...
bool b = true;
if (v.size() > 0) {
    auto itb = v.begin();
    auto ite = v.end() - 1;
    while (b && itb < ite) {
        b = (*itb) == (*ite);
        itb++;
        ite--;
    }
}

cout << (b ? "yes":"no");
return 0;
}
```

Обратный (реверсивный) итератор `v.rbegin()`

```
bool b = true;
auto itb = begin(v);
auto itr = rbegin(v);
while (b && itb < itr.base()) {
    b = (*itb) == (*itr);
    itb++;
    itr++;
}
cout << (b ? "yes":"no");
```

Использование обратного итератора **позволяет избежать дополнительной проверки на непустоту** вектора

Типы итераторов `itb` и `itr` — разные, поэтому сравнение для них неприменимо. Но **обратный** итератор можно **привести к базовому**, используя функцию `base()`

Объявления итераторов

равносильные объявления

```
auto itb = v.begin();  
auto itb = begin(v);
```

```
vector<int>::iterator itb = v.begin();  
vector<int>::iterator itb = begin(v);
```

равносильные объявления

```
auto itrb = v.rbegin();  
auto itrb = rbegin(v);
```

```
vector<int> :: reverse_iterator itrb = v.rbegin();  
vector<int> :: reverse_iterator itrb = rbegin(v);
```

Получение итераторов

До C++ 11

```
auto it = v.begin();
```

Начиная с C++ 11

```
auto it = begin(v)
```

Что даёт новый подход?

Пример функции печати каждого элемента контейнера

```
void print(const & c) {  
    for(auto it = begin(c), e = end(c); it != e; ++it)  
        std::cout << *it << ' '  
    std::cout <<std::endl;  
}
```

использование print

```
vector<int> v {1, 2, 3};  
int a[] = {4, 5, 6};  
print(v);  
print(a);
```

На самом деле эта функция **нарушает философию STL** так как в функцию **передается контейнер, а не итераторы.**

Есть ли проблемы?

Пример функции печати каждого элемента контейнера

```
void print(const & c) {  
    for(auto it = begin(c), e = end(c); it != e; ++it)  
        std::cout << *it << ' '  
    std::cout <<std::endl;  
}
```

использование print

```
vector<int> v {1, 2, 3};  
int a[] = {4, 5, 6};  
print(v);  
print(a);
```

Здесь функция print нормально отработает для вектора, но для массива она не должна отработать, так как **у массива нет begin() и end()**.

Однако для массива отработает следующая **специализация шаблона**:

```
template <class T, size_t N>  
T* begin(T(&arr)[N]) noexcept {  
    return arr;  
}
```

```
template <class T, size_t N>  
T* end(T(&arr)[N]) noexcept {  
    return arr + N;  
}
```

Собственные версии свободных begin/end

Допустим имеется тип `myvector` такой, что выполнены два условия:

- не имеет функций-членов `begin`, `end`
- мы не можем менять (добавить в него `begin / end`)

Тогда мы можем написать собственные версии свободных `begin/end` для этого типа, тогда `print` станет работать и с ним

Пример. Дан вектор. Удалить элементы вектора, начиная с последнего нуля. Необходимо, чтобы ёмкость вектора соответствовала количеству элементов.

```
#include <iostream>
#include <vector>
#include <algorithm>
...
int a[] = { 3,0,4,0,7 };
vector <int> v(a, a + 5);
vector<int> v0 {0};
cout << v.size() << " " << v.capacity() << endl;

auto it = find_end(v.begin(), v.end(), v0.begin(), v0.end());

v.erase(it, v.end());
cout << v.size() << " " << v.capacity() << endl;

vector<int> v1(v);
v.swap(v1);
cout << v.size() << " " << v.capacity() << endl;
```

5	5
3	5
3	3

Как изменить ёмкость у вектора

v.erase(it, v.end()); //удаление приводит к уменьшению размера, а не ёмкости

vector<int> v1(v); //создается копия по реальному количеству элементов

v.swap(v1); //обмен

Недействительный итератор

```
v.erase(it, v.end());  
//cout << *it << endl;    //ошибка времени выполнения
```

Для дальнейшего использования переменной `it` потребуется её повторная инициализация.

Рекомендации

Если предполагается **большое количество** операций **удаления/вставки** в **произвольных** местах контейнера, то рекомендуется использовать двунаправленный список **list** или однонаправленный **forward_list**.

Методы, работающие с указателями и диапазонами

`a.erase(pt)` удаление элемента по указателю `pt`

`a.erase(p1, p2)` удаление диапазона `[p1, p2)`

```
score.erase(begin(score), begin(score)+2);
```

`a.insert(p,t)` вставка значения `t` перед указателем `p`

`a.insert (pt, p1,p2)` вставка диапазона `[p1, p2)` из другого вектора перед итератором `pt`

```
vector<int> old;
```

```
vector<int> new;
```

```
...
```

```
old.insert(old.begin(),3);
```

```
old.insert(old.end(),5);
```

```
new.insert(new.begin(), old.begin(), old.end());
```

```
new.insert(new.end(), old.begin()+1, old.end()-1);
```

Пример. Дана строка, содержащая слова, разделённые запятыми, и завершающаяся точкой. После каждой запятой вставить пробел.

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    list<char> text;
    char c;
    do {
        cin >> c;
        text.push_back(c);
    } while (c != '.');
```

```
copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));

cout << endl;
auto ite = text.end();

auto itb = find(text.begin(),ite,',');
while (itb != ite) {
    itb=text.insert(++itb, ' ');
    itb = find(itb,ite,',');
}

copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));
}
```

Функция `insert(++itb, ' ')`

выполняет **вставку в позицию перед итератором**, поэтому необходимо сдвинуть итератор `++itb`

возвращает итератор на вставленный элемент

Алгоритм copy

```
copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));
```

параметрами которого являются итераторы на начало и конец копируемого контейнера и итератор на начало контейнера, куда производится копирование

для **вывода в поток** третьим параметром должен быть **итератор вывода в поток ostream**

```
ostream_iterator<char>(cout,""))
```

copy — вопросы

```
vector<int> v {3, 5, 2};  
list<int> l {2, 7, 8, 9};  
int a[10];  
auto pa = copy(v.begin(), v.end(), a);  
pa = copy(l.begin(), l.end(), pa);
```

Что будет, если в том контейнере, куда мы копируем, будет недостаточно места?

сору — вопросы и ответы

Что будет, если в том контейнере, куда мы копируем, будет недостаточно места?

Будет перезаписана чужая память из-за отсутствия контроля выхода за границу.

copy — ещё ошибки

```
vector<int> v{1, 5, 3};  
list<int> l;  
copy(begin(v), end(v), begin(l));
```

// Итератор будет иметь нулевое значение поэтому при попытке обратиться к нему произойдет ошибка.

Если написать так:

```
pa = copy(l.end(), l.begin(), pa);
```

Тогда copy от такой ошибки не застрахован

так же он не застрахован от ошибок в случае такого кода:

```
pa = copy(l.begin(), l1.end(), pa);
```