

# C++ ссылки (references)

## Декларация ссылок

Ссылка в C++ это альтернативное имя для уже существующего объекта (переменной) декларируется в виде:

`Type & RefName`

`Type` – тип переменной `int`, `double`, ...

`RefName` – имя ссылки (ссылочной переменной)

## Пример

```
int i = 1;           // обычная переменная i
int& ref_i = i;      // ref_i - ссылка на i
ref_i = 8;           // изменение i через ссылку
cout << i << endl;   // 8 печать i
```

## Важно помнить:

- ☞ Ссылка не может существовать сама по себе, она обязательно связана с уже существующей переменной (объектом):

```
int& ref_i = i;           // OK!  
int& ref_j;             // error: 'ref_j' not initialized  
int& ref_j = 1;         // error: can't bind a reference to a constant  
const int& ref_j = 1;    // reference to a constant anonymous variable
```

- ☞ Невозможно «перенаправить» ссылку на другую переменную:

```
int& ref_i = i;  
ref_i = j;           // меняем значение переменной 'i'!
```

- ☞ У ссылки нет своего адреса, адрес ссылки вернет адрес связанной с ней переменной:

```
cout << &i << " AND " << &ref_i << endl;  
// 0x7fffc3a1d5e8 AND 0x7fffc3a1d5e8
```

# Ссылки в аргументах функции

👉 Сравните передачу переменной в функцию **по ссылке** и **по адресу**

## ссылка как аргумент функции

```
void f_ref(int& x) {  
    x = 1;  
}  
  
int main () {  
    int z = 0;  
    f_ref(z); // ссылка на 'z'  
    cout << "f_ref: z= "  
          << z << endl;  
}
```

f\_ref: z= 1

## указатель как аргумент функции

```
void f_ptr(int* x) {  
    *x = 1;  
}  
  
int main () {  
    int z = 0;  
    f_ptr(&z); // адрес 'z'  
    cout << "f_ptr: z= "  
          << z << endl;  
}
```

f\_ptr: z= 1

## Пример: function swap() in C++ style with references

```
void swap(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    int i = 2;  
    int j = 40;  
    cout << "before: i= "<<i<<"", j= "<<j<<endl; // before: i= 2, j= 40  
    swap(i,j);  
    cout << "after: i= "<<i<<"", j= "<<j<<endl; // after: i= 40, j= 2  
}
```

# Функции возвращающие ссылки

```
int& fun(int& x) {  
    return x;  
}  
  
int z = 0;  
fun(z) = 2; // вызов функции слева!  
cout << "fun: z= " << z << endl; // fun: z= 2
```

## Зачем это нужно?

- Вызов функции теперь может стоять слева (L-value)
- Переопределение некоторых операторов **невозможно** без возврата ссылки
- Задание цепочек вызовов подобно: `cout << i << j << endl;`

Как работает цепочка `cout << i << j ...`

- 1 Запись `cout << i` означает вызов функции `operator<<(cout,i)`
- 2 Цепочка `cout << i << j`; разворачивается в `operator<<(operator<<(cout,i),j)`
- 3 Чтобы цепочка работала, функция `operator<<(cout,something)` должна вернуть ссылку на `cout`

## Перегрузка оператора вывода

```
ostream& operator << (ostream& out, const Rational& r) {  
    return out << r.x << "/" << r.y;  
}
```

- 👉 Первый аргумент и возвращаемое значение имеют тип `ostream&`
- 👉 Второй аргумент тоже ссылка: это «экономит» копирование при передаче класса

## Два значения символов \* и & в зависимости от контекста:

- 1 В декларациях являются частью идентификации типа:

```
int & ref_i = i; // декларируют ref_i как ссылку  
int * ptr_i = 0; // декларируют ptr_i как указатель
```

- 2 В выражениях это операторы:

```
ptr_i = &i; // взятия адреса переменной i  
j = *ptr_i; // косвенный доступ к данным по адресу в ptr_i
```

# Ссылки или указатели, что лучше?

👉 C++ FAQ: используйте ссылки где можете, а указатели только там где должны!

👉 В C++ ссылки на временные или динамические объекты порождают проблемы

## пример №1: возвращаем большой объект по ссылке

```
vector<int>& BadReferenceFun() {      // very bad function
    vector<int> v {1,2,3};
    return v;
}
vector<int>& vi = BadReferenceFun(); // reference is invalid
cerr << " vi[0]= " << vi[0] << endl; // vi[0]= Segmentation fault
```

👉 Проблема функций, возвращающих большие объекты, решается с помощью концепции перемещений (move) в C++11



## пример №2

```
vector<int>& ProblematicReferenceFun() { // questionable function
    vector<int>* v = new vector<int> {1,2,3};
    return *v;
}
```

```
vector<int>& rv = ProblematicReferenceFun(); // ok!
cerr << " rv[0]= " << rv[0] << endl;      // rv[0]= 1
delete &rv; // ok, but no 'NULL'-reference
cerr << " rv[1]= " << rv[1] << endl;      // rv[1]= 0?
```

```
vector<int> vv = ProblematicReferenceFun(); // memory leak!
```

## Так что же лучше?

👉 Пользуйтесь тем средством работу которого лучше понимаете

## Отсутствуют в C++:

### Массив переменной длины (Variable Length Arrays) в C99

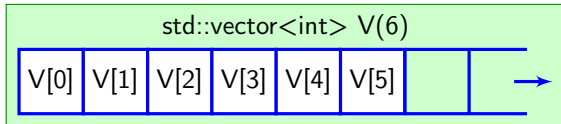
```
void fun(int n) {  
    int vla[n]; // ошибка для многих компиляторов C++  
    ...  
}
```

### В C++ вместо VLA используют `vector<>`

```
#include <vector>  
void fun(int n) {  
    vector<int> v(n);  
    ...  
}
```

# STL vector<>

Вектор (vector) – это *последовательный контейнер*



## Вектор как динамический массив

- Тип элементов указывается в угловых скобках:  
`vector<int>`, `vector<double>`, ...
- Подобно обычному массиву, вектор обеспечивает прямой доступ к произвольному элементу: `v[5] == *(&v[0]+5)`
- При изменении размера вектора (увеличении) происходит автоматическое перераспределение памяти

# STL vector: конструкторы

```
#include <vector>
using namespace std;
```

## Можно создать:

- Пустой вектор: `vector<int> v;`
- Вектор из N элементов «по умолчанию» или заданным значением  
`vector<Rational> R(50);` // 50 elements, default initialization  
`vector<double> d(10,1.);` // 10 elements, initialize with 1.
- Вектор копия другого вектора: `vector<double> c(d);`
- В C++11 можно инициализировать списком:  
`vector<int> v {1,2,3,4,5};`

# STL vector: доступ к элементам

<code>V[idx]</code>	элемент с индексом <code>idx</code> , как в обычном массиве
<code>V.front()</code>	первый элемент
<code>V.back()</code>	последний элемент
<code>V.at(idx)</code>	<code>== V[idx]</code> плюс проверка индекса (exception)

```
vector<int> v {1,2,3,4,5};
cout << " v[2]= " << v[2] << endl;           // v[2]= 3
cout << " v first element: " << v.front() << endl; // v first element: 1
cout << " v last element: " << v.back() << endl;  // v last element: 5
cerr << " v.at(5)= " << v.at(5) << endl;

v.at(5)= terminate called after throwing an instance of
'std::out_of_range' what(): vector::_M_range_check:
__n (which is 5) >= this->size() (which is 5)
```

# STL vector: размер и ёмкость

- Вектор содержит информацию о числе элементов

<code>V.size()</code>	количество элементов (размер) вектора
<code>V.empty()</code>	возвращает <code>true</code> для пустого вектора
<code>V.resize(num)</code>	делает число элементов равным <code>num</code> , удаляя или добавляя из/в конец вектора
<code>V.resize(num,elem)</code>	при добавлении используется <code>elem</code>

- Можно зарезервировать область памяти для использования

<code>V.reserve(num)</code>	резервирование памяти для <code>num</code> элементов
<code>V.capacity()</code>	размер зарезервированной памяти
<code>V.shrink_to_fit()</code>	уменьшает <code>capacity</code> до <code>size()</code> (C++11)

## STL vector: вставка и удаление

<code>V.push_back(elem)</code>	добавляет элемент в конец вектора
<code>V.pop_back()</code>	удаляет последний элемент
<code>V.insert(pos,elem)</code>	вставляет элемент перед итератором <code>pos</code> и возвращает итератор на него
<code>V.insert(pos,beg,end)</code>	вставляет перед итератором <code>pos</code> элементы из диапазона <code>[beg,end)</code>
<code>V.clear()</code>	удаляет все элементы
<code>V.erase(pos)</code>	удаляет элемент на позиции итератора <code>pos</code> и возвращает позицию следующего элемента
<code>V.erase(beg,end)</code>	удаляет все элементы в диапазоне <code>[beg,end)</code> и возвращает позицию следующего элемента

## Пример: размер и ёмкость

```
// 1) create vector of 5 elements
vector<int> vec {1,2,3,4,5}; // 1 2 3 4 5 : size= 5 capacity= 5

// 2) append one element
vec.push_back(-1);          // 1 2 3 4 5 -1 : size= 6 capacity= 10

// 3) remove the last element
vec.pop_back();             // 1 2 3 4 5 : size= 5 capacity= 10

// 4) remove all elements
vec.clear();                // : size= 0 capacity= 10

// 5) resize vector
vec.resize(4,0);            // 0 0 0 0 : size= 4 capacity= 10

// 6) shrink_to_fit
vec.shrink_to_fit();        // 0 0 0 0 : size= 4 capacity= 4

// 7) reserve memory
vec.reserve(25);            // 0 0 0 0 : size= 4 capacity= 25
```





# Передача вектора в функцию

**Пример: сумма элементов `vector<int>`**

```
int Sum(const vector<int> & m) { // use reference!
    int s = 0;
    for(unsigned int i = 0; i < m.size(); ++i) s += m[i];
    return s;
}

...
vector<int> V {5,4,3,2,1};
cout << " Sum(V)= " << Sum(V) << endl; // Sum(V)= 15
```

 используйте ссылку на `vector<>`

 константная ссылка гарантирует, что функция не меняет вектор

# Использование вектора как обычного массива

- Имеется функция для работы с обычными массивами типа `int`:

```
int Sum(int m[], int n) {  
    int s = 0;  
    for(int i = 0; i < n; ++i) s += m[i];  
    return s;  
}
```

Можно ли её использовать для `vector<int>`?

- ☞ Вектор всегда можно использовать вместо массива используя начало «внутреннего массива вектора»:

`VectorName.data()` (C++11) или `&VectorName[0]` (C++98)

```
vector<int> V {5,4,3,2,1};  
cout << " Sum[V]= " << Sum(V.data(),V.size()) << endl; // Sum[V]= 15  
// cout << " Sum[V]= " << Sum(&V[0],V.size()) << endl; // C++98
```

## Универсальная инициализация через фигурные скобки {}

```
double a = 1.2;           // обычная инициализация
double a {1.2};           // new C++11
double a = {1.2};         // в C++11 это новая инициализация!
char c[] {"abc"};         // инициализация массива
vector<int> v {1,2,3,4,5}; // вектора
```

## Универсальная инициализация более строга

```
int a = 1.2;    // a = 1 (warning in the best case)
int a {1.2};    // ERROR: 'double' cannot be narrowed to 'int'
char ch {332};  // ERROR: narrowing conversion
double d {a};   // warning: narrowing conversion 'int' to 'double'
```

## auto с инициализацией

☞ тип переменной определяется по типу правой части

```
auto a = 1.2;    // a - double
auto b = 1;      // b - int
auto c = true;   // c - bool
auto x = fun(c); // x - тип который возвращает fun(bool)
```

## перебор всех элементов вектора

```
vector<int> v {1,2,3,4};  
int s = 0;  
for(auto x : v) { // x - это копия элемента v[i]  
    s += x;  
}  
cout << " s= " << s << endl; // s= 10
```

## перебор с изменением элементов вектора

```
s = 0;  
for(auto& x : v) { // x - это ссылка на элемент v[i]  
    s += x;  
    x = s;          // сохраняем кумулятивную сумму  
}  
// v= 1 3 6 10
```

Часто используют следующий синтаксис:

- ❶ `for(auto x : v) {...}` // x - копия элемента вектора
- ❷ `for(auto& x : v) {...}` // x - ссылка на элемент
- ❸ `for(const auto& x : v) {...}` // x - константная ссылка

**Пример: «range-for» не только для вектора**

```
int a[] {1,6,8,9};  
for(const auto& x : a) { // an array  
    cout << x << " ";  
}  
cout << endl; // 1 6 8 9  
for(auto x : {1.,1.3,1.21}) { // braced-init-list  
    cout << x << " ";  
}  
cout << endl; // 1 1.3 1.21
```

- 1 Позволяет вычислить константы на стадии компиляции
- 2 Позволяет задать `constexpr` функцию, которую можно использовать как для вычисления констант компилятором, так и далее во время исполнения программы

## 1 Вычисление констант на стадии компиляции

```
constexpr unsigned long Factorial(unsigned int n) {  
    return n < 1 ? 1 : n*Factorial(n-1); // no loops in C++11  
}  
  
constexpr int F10 = Factorial(10); // compile time  
constexpr double half_ln_2pi=log(2*M_PI)/2; // is std::log() constexpr?  
constexpr array v {1,2,3}; // C++17  
constexpr auto sum = v.front() + v.back(); // C++17
```

## 2 Использование constexpr функции

```
constexpr unsigned long Factorial14(unsigned int n) { // C++14
    unsigned long result = 1UL;
    for(; n>1; --n) result *=n;
    return result;
}

constexpr int F7 = Factorial14(7);                // compile time
cout << "F7= " << F7 << endl;                    // F7= 5040
volatile int b = 8;                               // disallow optimization
cout << b << "!=" << Factorial14(b) << endl; // 8!=40320 - run time
```

📌 constexpr подразумевает const

📌 constexpr функция может вызываться как обычная функция

📌 вычисления должны быть *«достаточно простыми»*