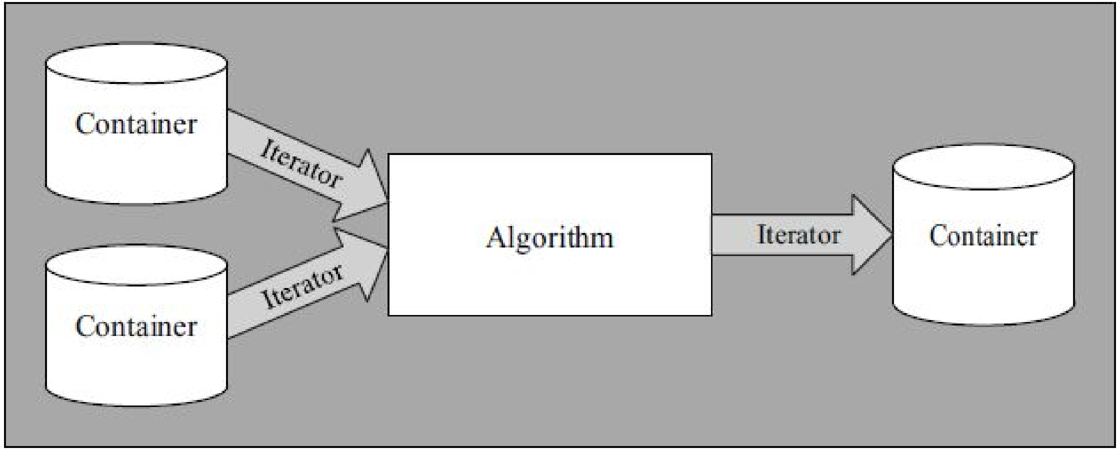
C++ Course 7: STL. Containers. Iterators. Algorithms.

By Oleksiy Grechnyev

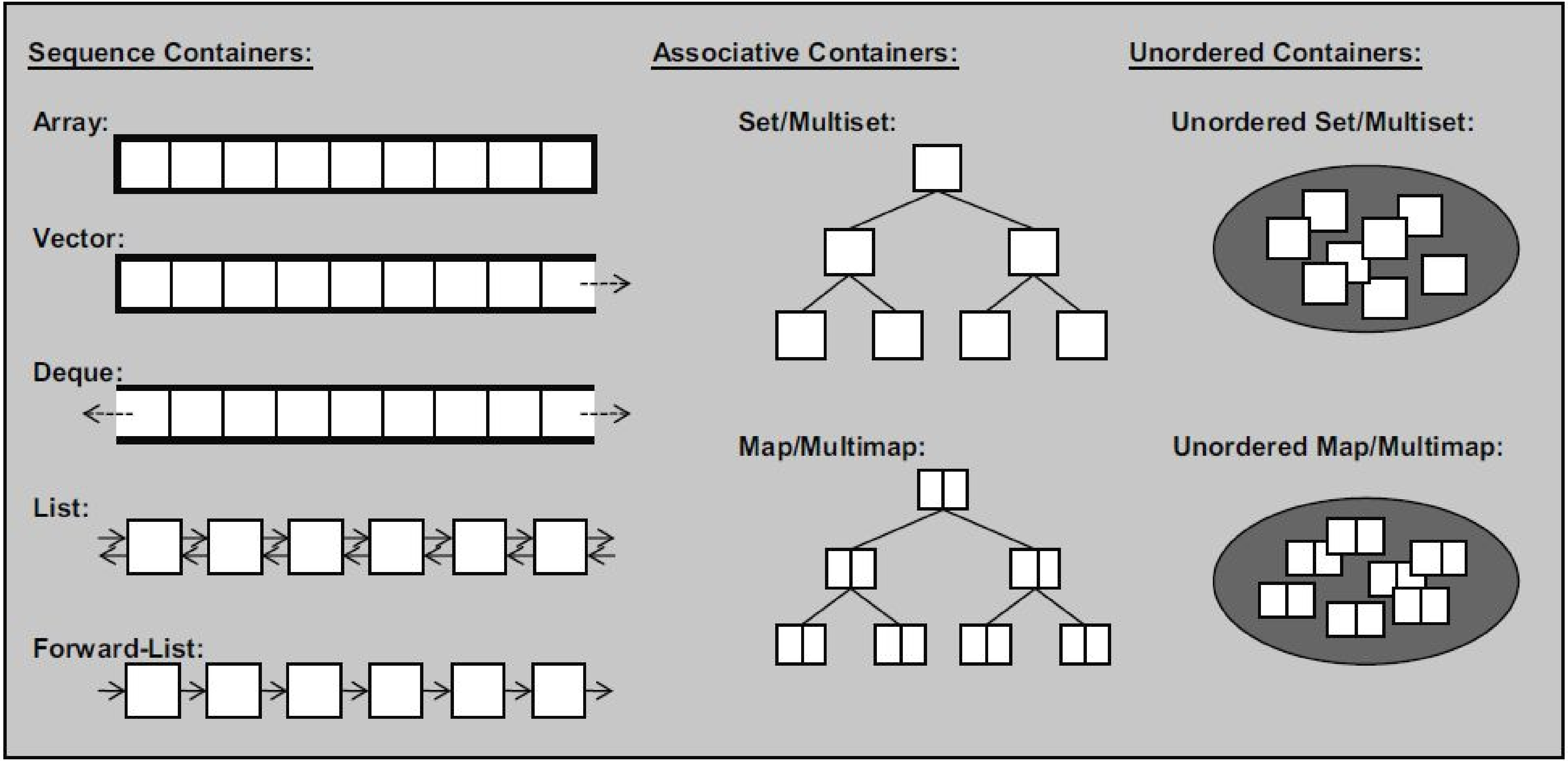
Standard Template Library (STL)



1. **Container** : A data class : **vector, array, map, set,** ...
2. **Iterator** : An object which iterates over a container (sort of a smart pointer)
3. **Algorithm**: A polymorphic algorithm : **sort, find, reverse, transform, copy, move** ...
4. **Function Objects + Lambda expressions**: Often used as arguments in algorithms

STL = Object Oriented + Generic + Functional programming

STL containers



Built-in arrays

```
int a[12];
double b[3] = {0.1, 1.2, 2.3};
string weapons[] = {"Sword", "Axe", "Bow"};
constexpr int SIZE = 1024*1024*16;    // SIZE must be constexpr
char buffer[SIZE];
char cString[] = "This is a null-terminated C-string";   // Automatically ended with \0
double matrix[10][10];      // Multidimensional
```

Arrays are indexed by the [] operator, starting from 0, no range checks !

```
for (int i=0; i<12; ++i)
        a[i] = i*i;
```

Arrays can be converted to pointers. Преобразование в указатели.

```
char * message = cString;
```

Size of an array (number of elements):

```
int size = sizeof a / sizeof (int);
```

References and Pointers to array

Do not confuse with pointer to array *element*. Array type must be of fixed size !

```
int a[12];
int (& aRef1) [12] = a;
int (* aPtr1) [12] = &a;
```

Create a type alias: Создайте новый тип.

```
using ArrayType = int [12];
ArrayType & aRef2 = a;
ArrayType * aPtr2 = &a;
```

Template to get array size:

```
template <typename T, size_t SIZE>
size_t getArraySize(const T (&) [SIZE]) { return SIZE; }
```

Arrays can be printed with a range for:

```
for (int i : a) cout << i << " ";
```

Dynamic arrays (pointers actually)

How do we create an array of dynamic size?

We cannot, use pointers instead:

```
int size = 1024;                // Not constexpr
int * data = new int[size];
for (int i=0; i<size; ++i)
      data[i] = i*i;            // We can use operator[] with pointers
delete [] data;                 // Array delete
```

It is possible to use **unique_ptr** (but not **shared_ptr** !) :

```
unique_ptr<int[]> data2(new int[size]);
```

Pointers are not real arrays!

You cannot use range **for, begin(), end(),** or our **getArraySize()** for pointers !

Absolutely no way to tell the size !

sizeof(data) returns pointer size (8 bytes) and not array size !

Pointers as array iterators

A C-style Array (NOT class) :

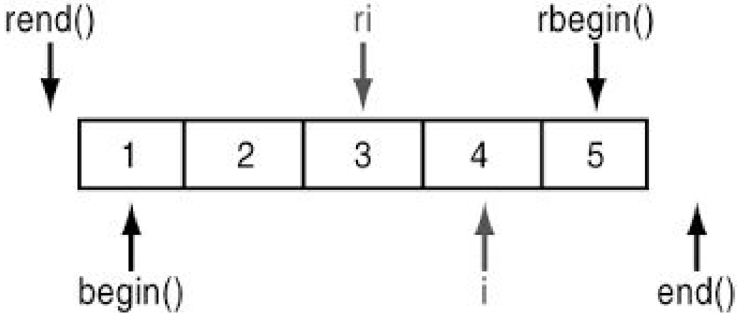
```
const string names[] = {"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
```

Print it with pointers :

```
const string * eit = names + 5;    // Position just after the last element
for (const string * it = names; it != eit; ++it)
     cout << *it << " ";
```

Or using C++ iterator style (only works with real array, not pointers !):

```
for (const auto *it = begin(names); it != end(names); ++it)
     cout << *it << " ";
```



What's wrong with built-in arrays? C legacy.

1. They cannot be copied. Нельзя копировать.
2. They cannot be returned from a function. Нельзя возвращать.
3. They cannot be passed to a function by value. Converted to pointer !!!

```
void fun(int a[]) {...}          // Converted to int * pointer !
void fun(int a[37]) {...}        // Converted to int * pointer, size is ignored !
```

4. No **size()** method !
5. Must be of fixed size.
6. Array types in templates and containers are trouble.
7. Pointer is NOT an array. No range **for, begin(), end()** for pointers !

Don't use arrays, use *container classes* instead !

Не используйте массивы, используйте контейнеры !

Exception: Array of constants: Исключение: Массив констант:

```
const string names[] = {"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
```

std::array : Array of fixed size

```
array<string, 5> aS1{"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
array<int, 100> aI;
aI.fill(17);                                  // Fill with the value 17
constexpr int SIZE = 1024*1024*16;
array<double, SIZE> aD;          // Size must be constexpr !
auto aStr = std::experimental::make_array("Red", "Green", "Blue");
```

Possible implementation of **array**:

```
template <typename T, size_t SIZE>
class array{
public:
        ...                              // Methods, operators
private:
        T myData[SIZE];     // The build-in array
};
```

Creating std::array : more options

You can use type alias:

```
using SArray = array<string, 5>;
SArray aS1{"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
SArray aS2 = {"Maria", "Nel", "Sophia", "Clair", "Mirage"};  // This is also OK
SArray aS3;
aS3 = aS1;                  // Copy array
aS1.swap(aS2);          // Swap arrays
swap(aS1, aS2);          // Swap arrays (the same)
```

Get a raw pointer to the data (underlying built-in array):

```
string * rawData = aS1.data();
```

Create an **std::array** object out of a built-in array (NOT from pointer !):

```
string a[] = {"Maria", "Nel", "Sophia", "Clair", "Mirage"};
auto aS4 = std::experimental::to_array(a);
```

std::array of funny types

Pointers (but NOT references) :

```
int i1 = 13, i2 = 17, i3 = 666;
array <int *, 3> aPtr{&i1, &i2, &i3};           // Pointers
array <const int *, 3> aCPtr{&i1, &i2, &i3};    // Pointers to const
```

Constants:

```
array<const string, 5> cNames{"Maria", "Nel", "Sophia", "Clair", "Mirage"};
```

unique_ptr or **shared_ptr** objects:

```
array<unique_ptr<int>, 2> uAr {
        make_unique<int>(17),
        make_unique<int>(666)
};
```

Built-in arrays:

```
array<int[17], 3> aa;
```

Indexing std::array

```
array <int, 12> a;
```

a.at(i)	: Element **i** (Checks boundaries, throws **std::out_of_range**)
a[i]	: Element **i** (No checks)
a.front()	: First element
a.back()	: Last element
a.size()	: Number of elements

Set/modify array elements:

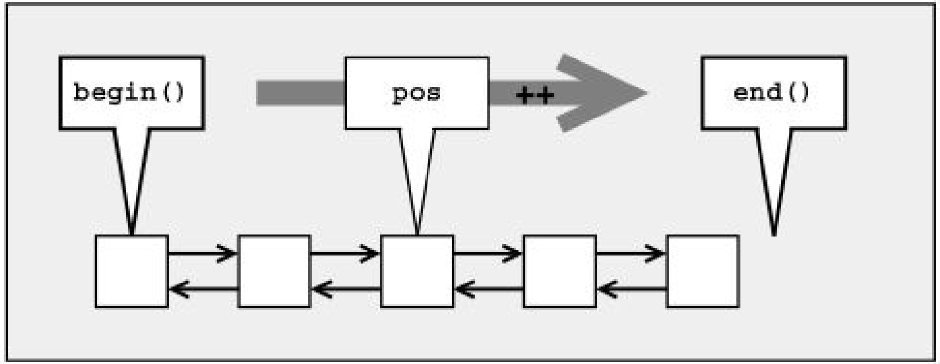
```
for (int i = 0; i < a.size(); ++i)
        a.at(i) = i*i;
```

Print the array:

```
for (int i = 0; i < a.size(); ++i)        // Using []
        cout << a[i] << " ";
for (int elem : a)                        // Using range for
        cout << elem << " ";
```

Iterators

- *Iterators* are objects (smart pointers) that iterate over container elements. Итератор пробегает элементы контейнера.
- This includes containers without numerical index (**std::set**, **std::list**). Это включает контейнеры без числового индекса.
- Iterators have operators ***** and **->** defined (like pointers). ***iter** is the container element the iterator points at (элемент контейнера на который указывает итератор).
- All iterators support operators **++** , **==** , **!=**, **=**.
- Some iterators support operators **--** , **+**, **-** , **<**, **>**, **+=**, **-=**.

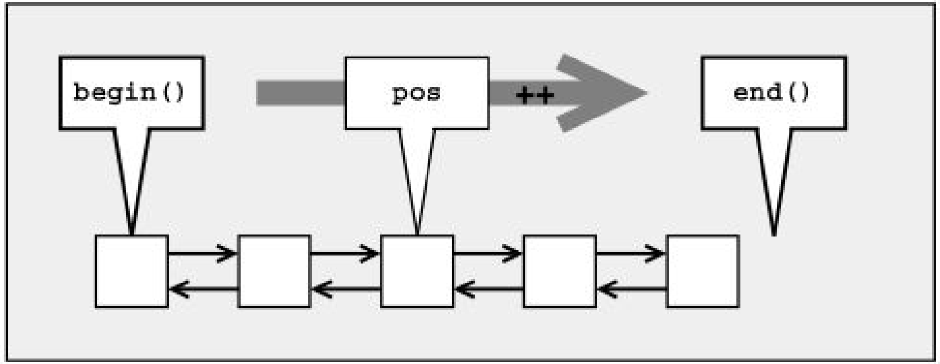


begin() and end()

a.begin() or **begin(a)** is the iterator to the first element of a container **a**.
a.begin() или **begin(a)** -- итератор на первый элемент контейнера **a**.

a.end() or **end(a)** is the iterator *past the last* element of a container **a**.
a.end() or **end(a)** -- итератор после последнего элемента контейнера **a**.

a.end() is not a valid element!
a.begin() == a.end() for an empty container.



const and reverse iterators, for loop

Normal version : **a.begin()**, **a.end()**

const version : **a.cbegin()**, **a.cend()**

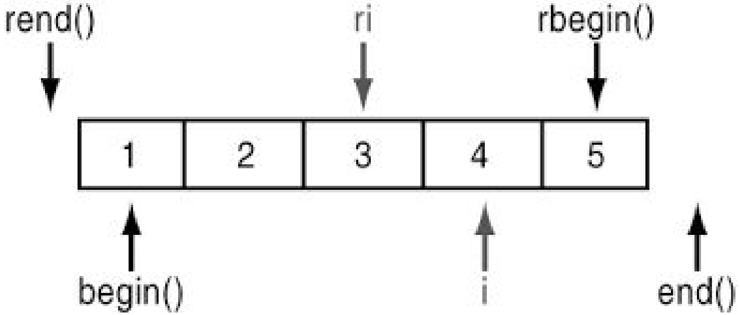
Reverse version : **a.rbegin()**, **a.rend()** all except **forward_list**

Reverse **const** version : **a.crbegin()**, **a.crend()** all except **forward_list**

Iterators in a **for** loop:

```
for (auto it = a.begin(); it != a.end(); ++it)
        *it *= *it;   // Square each element of the container
```

Range **for** is based on iterators!



Iterator arithmetics (std::array, std::vector only)

Index to iterator:

```
auto it = a.begin() + index;
```

Iterator to index:

```
size_t index  = it - a.begin();
```

More operations:

```
int diff  = it2 - it1;         // Distance between two iterators
if (it1 < it2) ..              //  Compare two iterators
it += 5;                        // Move forward by 5 elements
it -=  2;                       // Move back by 2 elements
it =  a.end() - 1;              // Element before last
```

Iterator classes of **std::array** :

array::iterator,   array::const_iterator,

array::reverse_iterator,   array::const_reverse_iterator

Templates to print any container (even built-in array !)

With range **for**:

```
template <typename C>
void print(const C & c){
    for (const auto & e : c)
        cout << e << " ";
    cout << endl;
}
```

With iterators:

```
template <typename C>
void print2(const C & c){
    for (auto it = begin(c); it != end(c); ++it)
        cout << *it << " ";
    cout << endl;
}
```

Iterators, ranges, and algorithms

C++ algorithms use a range given by 2 iterators : **first**, **last**.

C++ алгоритмы используют диапазон заданный 2 итераторами : **first**, **last**.

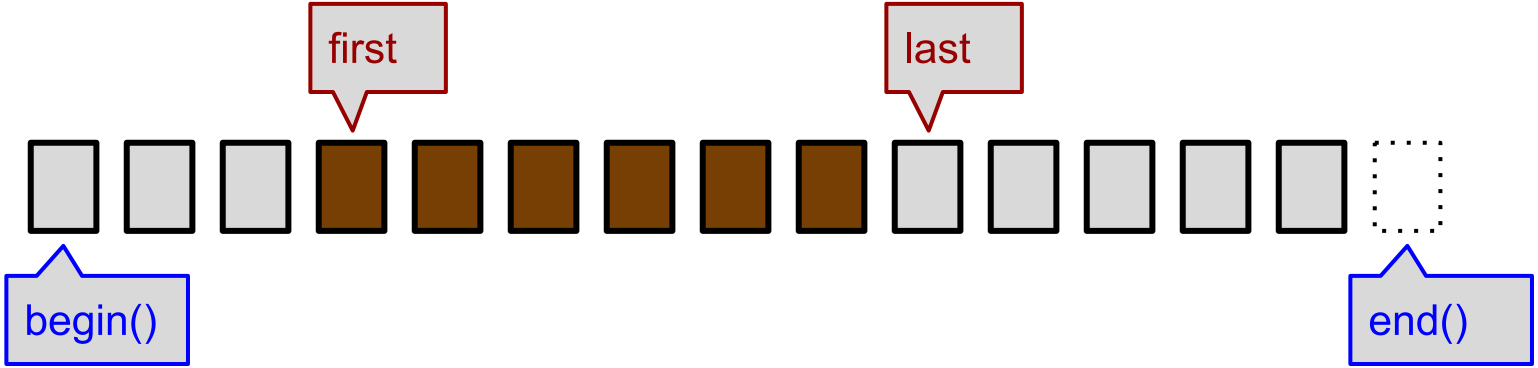
first : The first element of the range (Первый элемент диапазона)

last : The element past the last (Элемент после последнего)

Use **begin()**, **end()** to include the entire container in the range.

Используйте **begin()**, **end()** чтобы включить весь контейнер в диапазон.

The algorithms are *polymorphic templates* (work with any container !)



Algorithms 1

Sort a container. Сортировка.

```
sort(first, last);
sort(a.begin(), a.end());       // The entire container
sort(begin(a), end(a));         // This form can be used for built-in arrays
```

Reverse a container. Изменить порядок на обратный.

```
reverse(first, last);
```

Random order. Случайный порядок.

```
shuffle(first, last, rng);      // rng = Random Number Generator
```

Find min and max elements (Returns iterator !).

```
auto minEl = min_element(first, last);
auto maxEl = max_element(first, last);
cout << "min = " << *minEl << ", max = " << *maxEl << endl;
```

Algorithms 2

Fill with a value. Заполнить значением. : **fill(first, last, value);**

fill(a.begin(), a.end(), 13); // Fill container **a** with value 13

Copy elements. Копировать элементы. : **copy(first, last, dest_first);**

copy(a1.begin(), a1.end(), a2.begin()); // Copies **a1** to **a2**

Generate with a function. Генерировать с помощью функции: **generate(first, last, fun);**

int n = 0;

generate(al4.begin(), al4.end(), [&n](){return n++;}); // 0, 1, .., 11

Apply a function to each element: **for_each(first, last, fun);**

for_each(a.begin(), a.end(), [](int &n){n*=3;}); // Multiply each element by 3

Algorithms 3

Find first occurrence of an element. Returns last if not found.

```
find(first, last, value);
```

Find example:

```
array<int, 12> a{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121};
it1 = find(a.begin(), a.end(), 16);
it2 = find(a.begin(), a.end(), 64);

if (it1 == a.end() || it2 == a.end())    // Check if found
        throw runtime_error("Not found !!!");

if (it1 > it2)              // Check the order !
                swap(it1, it2);
reverse(it1, it2 + 1);    // Reverse from 16 to 64 inclusive !
```

Dangers of using iterators

When using algorithms, **last** must be reachable from **first** by operator **++** !

last должен быть достижим из **first** операцией **++** !

sort(first, last);

For example:

++ ++ ++ ++ ++ first == last

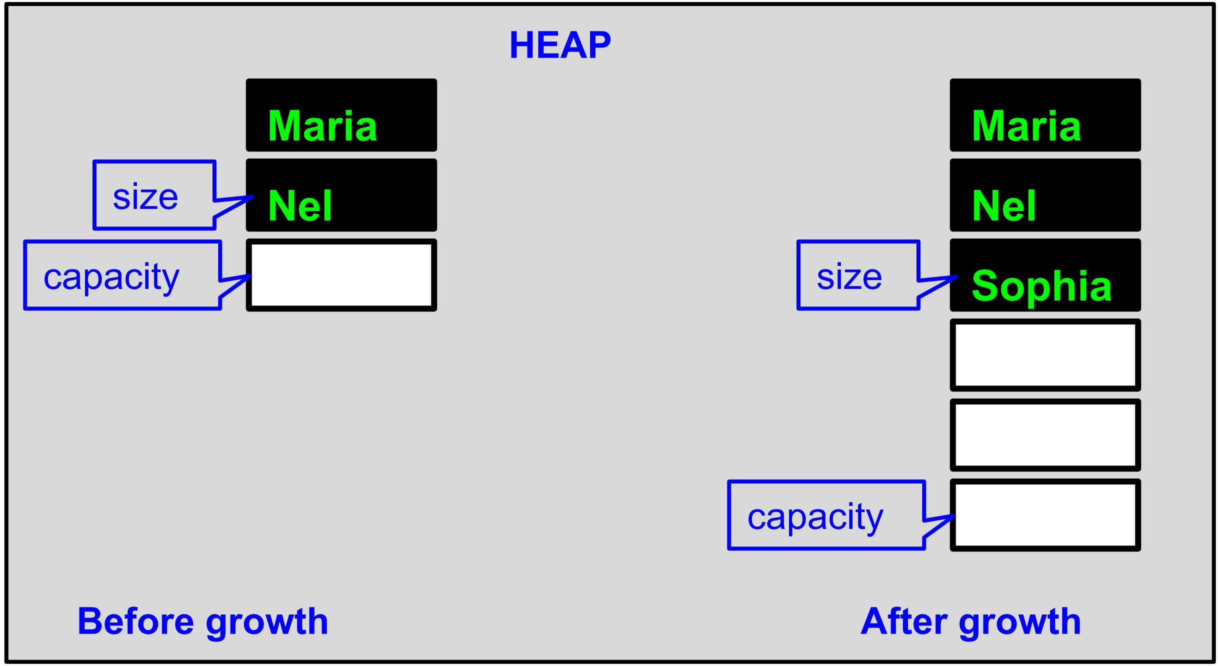
Otherwise BAD error!

```
array<int, 12> a1, a2;
...
sort(a1.begin(), a1.end());          //  OK
sort(a1.end(), a1.begin());          //  Error ! Wrong order !
sort(a1.begin(), a2.end());          //  Error ! Iterators of different arrays !
```

std::vector : A Dynamic array

```
vector<string> vS;
```

string ***data**;
int **size**;
int **capacity**;



size Number of objects in container

capacity Number of reserved slots in the heap

Creating std::vector

```
vector<int> vl1;                 // Empty vector
vl1.push_back(17);               // Add elements to an empty vector
vl1.push_back(19);
vl1.push_back(26);

vector<int> vl2(10);             // vector of 10 elements

vector<int> vl3(10, 13);         // vector of 10 elements equal to 13

vector<int> vl4{10, 13};         // vector of two elements : 10, 13

vector<int> vl5 = {2, 3, 7, 11, 13, 17, 19, 23};     // List assignment constructor

vector<string> vS{"Maria", "Nel", "Sophia", "Clair", "Mirage"};     // List constructor
```

move() and **swap()** are very good for vectors !

Filling std::vector with data

Indexing : Default Ctor, string Ctor, move assignment :

```
vector<Tjej> vT(5);                   // Default constructor 5 times !!!
for (int i=0; i < 5; ++i)
    vT.at(i) = Tjej("Tjej #" + to_string(i) );
```

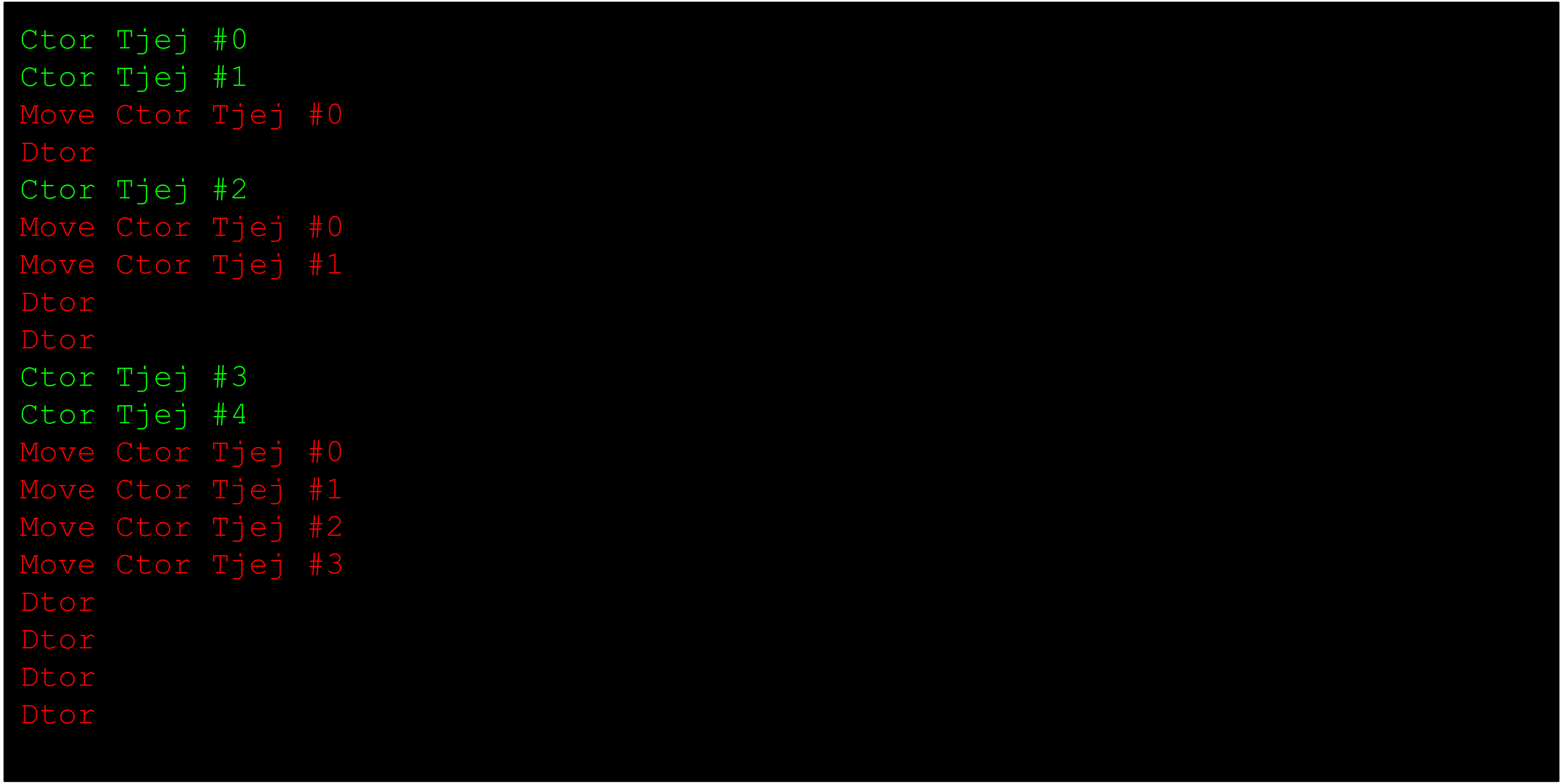
push_back : string Ctor, move Ctor :

```
vector<Tjej> vT;                       // Empty vector
for (int i=0; i < 5; ++i)
    vT.push_back(Tjej("Tjej #" + to_string(i) ));
```

emplace_back : string Ctor. New objects are constructed in-place !

```
vector<Tjej> vT;                       // Empty vector
for (int i=0; i < 5; ++i)
    vT.emplace_back("Tjej #" + to_string(i) );
```

But what the hell is going on ???



```
Ctor Tjej #0
Ctor Tjej #1
Move Ctor Tjej #0
Dtor
Ctor Tjej #2
Move Ctor Tjej #0
Move Ctor Tjej #1
Dtor
Dtor
Ctor Tjej #3
Ctor Tjej #4
Move Ctor Tjej #0
Move Ctor Tjej #1
Move Ctor Tjej #2
Move Ctor Tjej #3
Dtor
Dtor
Dtor
Dtor
```

std::vector capacity growth

```
vector<int> v;
for (int i = 0; i <= 40; ++i) {
        cout << "size = " << v.size() << ", capacity = " << v.capacity() << endl;
        v.push_back(i);
}
```

```
size = 0, capacity = 0
size = 1, capacity = 1
size = 2, capacity = 2
size = 3, capacity = 4
size = 4, capacity = 4
size = 5, capacity = 8
...
size = 9, capacity = 16
...
size = 17, capacity = 32
...
size = 33, capacity = 64
...
```

size vs capacity

SIZE operations:

```
v.size();                   // Get size
v.clear();                  // Delete all elements
v.resize(17);               // Change size (delete elements or create empty ones)
```

CAPACITY operations:

```
v.capacity();               // Get capacity
v.reserve(1000);            // Reserve storage (grow in size to given capacity)
v.shrink_to_fit();          // Trim capacity to size
```

Use **reserve()** before **push_back** / **emplace_back** !

Is growth a COPY or MOVE operation ?
If move constructor is **noexcept** : prefer MOVE !
Otherwise prefer COPY !
Don't forget **noexcept** in your move Ctor!

insert()/emplace() in the middle of vector

```
vector<int> v{1, 2, 3, 4, 5};

auto pos = find(v.cbegin(), v.cend(), 3);  // Find position of 3
pos = v.insert(pos, 17);                   // Insert BEFORE 3
pos = v.insert(pos, {21, 22});             // Insert list before 17
pos = v.emplace(pos, 33);                  // Emplace BEFORE 21
// insert() returns iterator to the 1st new element !

// 1 2 33 21 22 17 3 4 5

for (auto it = v.cbegin(); it != v.cend(); ++it)
    if (*it > 20 && *it < 30)
        it = v.insert(it, 49) + 1;         // Insert 49 BEFORE 21, 22
// 1 2 33 49 21 49 22 17 3 4 5

// it -> 21
// insert 17 before 21, it -> 17
// +1 : it -> 21
// ++it : iy -> 22
```

erase() : delete elements

```
vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

auto pos = find(v.cbegin(), v.cend(), 2);      // Find 2
pos = v.erase(pos);                           // Erase 2, pos -> 3
pos = v.erase(pos, pos+3);                    // Erase 3, 5, 6
// erase() returns iterator to the element AFTER erased

// 0 1 6 7 8 9

v.assign({0, 1, 2, 3, 4, 5, 6, 7, 8, 9});     // Just like Ctor

// Delete all even numbers in a for loop
for (auto it = v.cbegin(); it != v.cend(); ++it)
    if (*it % 2 == 0)
        it = v.erase(it) - 1;                 // -1 to negate ++it

// 1 3 5 7 9
```

set, unordered_set, multiset, unordered_multiset

std::set : Tree set. Sorted. Uses **operator<** or **less()** to compare.

std::unordered_set : Hash set. Unsorted. Uses function **hash()** .

std::multiset, std::unordered_multiset : Multisets can keep multiple copies.

```
set<int> s{1, 22, 2, 3, 19, 1, 3, 8, 12, 19, 22};        // Repeated, unsorted
```

Contains : 1 2 3 8 12 19 22

Look for element in the set:

```
int i = s.count(22);          // Returns 0 or 1 (or number of copies)
auto pos = s.find(22);     // Returns iterator or s.end() if not found
```

Insert and delete:

```
s.insert(5);
s.emplace(7);
s.insert({11, 13, 17, 19, 23});
s.erase(8);
```

std::pair : pair of objects of (possibly) different types

```
template <typename T, typename U>
struct pair{
        ...
        T first;
        U second;
};

pair<string, int> p1("Maria Traydor", 19);
auto p2 = make_pair("Nel Zelpher", 23);
pair<string, int> p3;
p3 = {"Sophia Esteed", 19};

cout << p1.first << " : " << p1.second << endl;
cout << p2.first << " : " << p2.second << endl;
cout << p3.first << " : " << p3.second << endl;
```

map, unordered_map, multimap, unordered_multimap

map<K, V> is a container of **pair<const K, V>** :

```
map<string, int> m1{
        {"Maria Traydor", 19},
        {"Nel Zelpher", 23}
};
m1.insert({"Mirage Koas", 27});                  // Create a new entry
m1.insert(make_pair("Sophia Esteed", 19));
m1.emplace("Peppita Rossetti", 14);
m1["Clair Lasbard"] = 25;                        // Create OR change

m1["Mirage Koas"] -= 1;                          // Create OR change
m1.at("Nel Zelpher") -= 1;                       // Change only
for (auto & p : m1)                              // Change with a range for
      p.second += 1;                             // p is pair<const string, int>
```

Templates to print a map

With range **for** :

```
template <typename M>
void printMap(const M & m){
    for (const auto & e : m)
        cout << e.first << " : " << e.second << endl;
}
```

With iterators:

```
template <typename M>
void printMap2(const M & m){
    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << " : " << it->second << endl;
}
```

map operations

Check for an entry

```
m1.count("Nel Zelpher");                // Returns 0 or 1
```

Find an entry (returns iterator)

```
auto pos = m1.find("Nel Zelpher");
```

Delete an entry (by iterator)

```
m1.erase(pos);
```

Delete an entry (by key)

```
m1.erase("Mirage Koas");
```

Number of entries

```
m1.size();
```

Thank you for your attention !

title

text