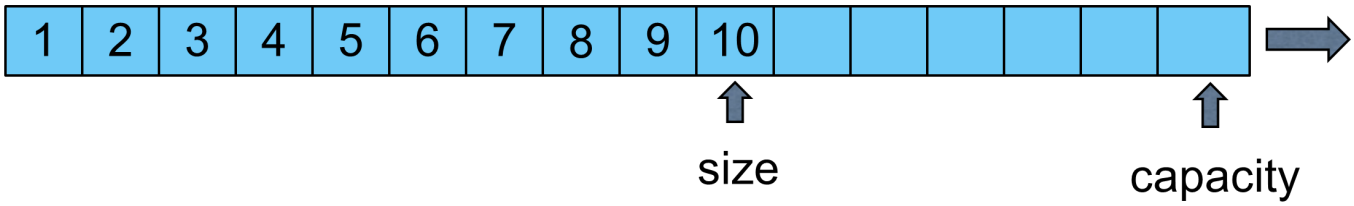
Vectors

A more refined version of arrays, vectors simplify insertion and deletion of values.

WE'LL COVER THE FOLLOWING

- Size versus Capacity



std::vector is a homogeneous container, for which it's length can be adjusted at runtime. `std::vector` needs the header `<vector>`. As it stores its elements contiguously in memory, `std::vector` support pointer arithmetic.

```
for (int i= 0; i < vec.size(); ++i){
  std::cout << vec[i] == *(vec + i) << std::endl; // true
}
```

Distinguish the round and curly braces by the creation of a `std::vector`

If you construct a `std::vector`, you have to keep a few specialities in mind. The constructor with round braces in the following example creates a `std::vector` with 10 elements, the constructor with curly braces a `std::vector` with the element 10.

```
std::vector<int> vec(10);
std::vector<int> vec{10};
```

The same rules hold for the expressions `std::vector<int>(10, 2011)` or `std::vector<int>{10, 2011}`. In the first case, you get a `std::vector` with

> 10 elements, initialised to 2011. In the second case, you get a `std::vector` with the elements 10 and 2011. The reason for the behaviour is, that curly braces are interpreted as initialiser lists and so, the sequence constructor is used.

Size versus Capacity

The number of elements a `std::vector` has is usually smaller than the number of elements for which is space is already reserved. That is for a simple reason. The size of the `std::vector` can increase without an expensive allocation of new memory.

There are a few methods for the smart handling of memory:

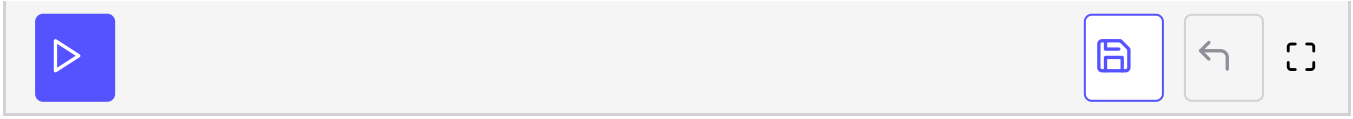
Method	Description
`vec.size()`	Number of elements of `vec`.
`vec.capacity()`	Number of elements, which `vec` can have without reallocation.
`vec.resize(n)`	`vec` will be increased to n elements. `
`vec.reserve(n)`	Reserve memory for at least n elements.
`vec.shrink_to_fit()`	Reduces capacity of `vec` to the size.

Memory management of `std::vector`

The call `vec.shrink_to_fit()` is not binding. That means the runtime can ignore it. But on the popular platforms, I always observed the desired behaviour.

So let's see the methods in the application.

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

int main(){
  std::vector<int> intVec1(5, 2011);
  intVec1.reserve(10);
  std::cout << intVec1.size() << std::endl;     // 5
  std::cout << intVec1.capacity() << std::endl; // 10

  intVec1.shrink_to_fit();
  std::cout << intVec1.capacity() << std::endl; // 5

  std::vector<int> intVec2(10);
  std::cout << intVec2.size() << std::endl;     // 10

  std::vector<int> intVec3{10};
  std::cout << intVec3.size() << std::endl;     // 1

  std::vector<int> intVec4{5, 2011};
  std::cout << intVec4.size() << std::endl;     // 2
  return 0;
}
```



std::vector

`std::vector vec` has a few methods to access its elements. With `vec.front()` you get the first element, with `vec.back()` you get the last element of `vec`. To read or write the (n+1)-th element of `vec`, you can use the index operator `vec[n]` or the method `vec.at(n)`. The second one checks the boundaries of `vec`, so that you eventually get a `std::range_error` exception.

Besides the index operator, `std::vector` offers additional methods to assign, insert, create or remove elements. See the following overview.

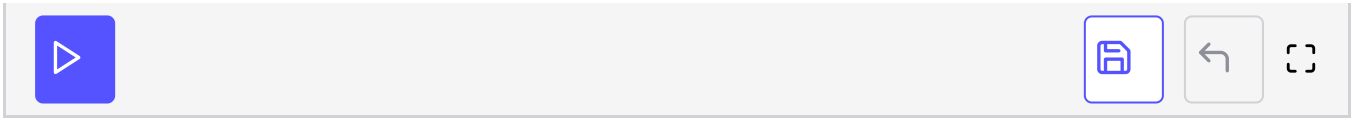
Method	Description
`vec.assign( ... )`	Assigns one or more elements, a range or an initialiser list.
`vec.clear()`	Removes all elements from `vec`.
`vec.emplace(pos, args ... )`	Creates a new element before `pos` with the `args` in `vec` and

	returns the new position of the element.
`vec.emplace_back(args ... )`	Creates a new element in `vec` with `args ...` .
`vec.erase( ... )`	Removes one element or a range and returns the next position.
`vec.insert(pos, ... )`	Inserts one or more elements, a range or an initialiser list and returns the new position of the element.
`vec.pop_back()`	Removes the last element.
`vec.push_back(elem)`	Adds a copy of `elem` at the end of `vec` .

Modify the elements of a `std::vector`

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

void display(std::vector<int> &v)
{
    for(auto i : v)
    {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}

int main(){
  std::vector<int> intVec1;
  intVec1.push_back(10);
  display(intVec1); // 1

  intVec1.push_back(20);
  intVec1.push_back(40);
  display(intVec1); // 1 2 4

  intVec1.insert(intVec1.begin() + 2, 30); // Inserting 30 at the 2nd index
  intVec1.insert(intVec1.begin(), 0); // Inserting 0 at the beginning
  display(intVec1); // 0 1 2 3 4

  intVec1.erase(intVec1.begin() + 1); // Deleting 10
```

```
  display(intVec1); // 0 20 30 40

  intVec1.pop_back();
  display(intVec1); // 0 20 30

  intVec1.assign(4, 100); // Replacing the elements with four 100s
  display(intVec1); // 100 100 100 100

  intVec1.clear();
  display(intVec1); // Empty

  return 0;
}
```



The `begin()` method returns an iterator pointing to the first element. Adding an integer value to it will move the iterator forward by that value. `begin()` is required to specify `pos` in methods like `insert` and `emplace`.