Arrays and Vectors

Victor Eijkhout, Susan Lindsey

Fall 2025

`last formatted: September 18, 2025`

1. Short vectors

Short vectors can be created by enumerating their elements:

```
// array/shortvector.cpp
#include <vector>
using std::vector;

int main() {
  vector<int> evens{0,2,4,6,8};
  vector<float> halves = {0.5, 1.5, 2.5};
  auto halfloats = {0.5f, 1.5f, 2.5f};
  cout << evens.at(0)
       << " from " << evens.size()
       << '\n';
  return 0;
}
```

2. Range over elements

A range-based for loop gives you directly the element values:

```
vector<float> my_data(N);
/* set the elements somehow */;
for ( float e : my_data )
  // statement about element e
```

Here there are no indices because you don't need them.

3. Range over elements, version 2

Same with auto instead of an explicit type for the elements:

```
for ( auto e : my_data )
  // same, with type deduced by compiler
```

4. Range over elements

Finding the maximum element

Code:

```cpp
// array/dynamicmax.cpp
vector<int> numbers = {1,4,2,6,5};
int tmp_max = -2000000000;
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max
     << " (should be 6)" << '\n';
```

Output:

```
Max: 6 (should be 6)
```

Exercise 1

Indicate for each of the following vector operations whether you prefer to use an indexed loop or a range-based loop. Give a short motivation.

- Count how many elements of a vector are zero.
- Find the location of the last zero.

5. Range over vector denotation

Code:

```
// array/rangedenote.cpp
for ( auto i : {2,3,5,7,9} )
  print( "{},",i );
println();
```

Output:

```
2,3,5,7,9,
```

6. Vector definition

Definition and/or initialization:

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

vector<type> name;
vector<type> name(size);
vector<type> name(size,init_value);
```

where

- `vector` is a keyword,
- *`type`* (in angle brackets) is any elementary type or class name,
- *`name`* of the vector is up to you, and
- `size` is the (initial size of the vector). This is an integer, or more precisely, a `size_t` parameter.
- Initialize all elements to *`init_value`*.
- If no default given, zero is used for numeric types.

7. Accessing vector elements

Square bracket notation (zero-based):

Code:

```
// array/assign.cpp
vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

Output:

```
4,8
```

8. Accessing vector elements

With bound checking:

Code:

```
// array/assign.cpp
vector<int> numbers = {1,4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << '\n';
```

Output:

```
4,8
```

Safer, slower.
(Remember Knuth about optimization.)

9. Vector elements out of bounds

Square bracket notation:

Code:

```
// array/assignoutofbound.cpp
vector<int> foo(25);
vector<int> numbers = {1,4};
numbers[-1] += 3;
numbers[2] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

Output:

```
1,4
```

10. Vector elements out of bounds

With bound checking:

Code:

```
// array/assignoutofbound.cpp
vector<int> numbers = {1,4};
numbers.at(-1) += 3;
numbers.at(2) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << '\n';
```

Output:

```
libc++abi:
      ↪terminating
      ↪with uncaught
      ↪exception of
      ↪type
      ↪std::out_of_range:
      ↪vector
```

Safer, slower.
(Remember Knuth about optimization.)

11. Range over elements by reference

Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
// array/vectorrangeref.cpp
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
println( "{}",myvector.at(2) );
```

Output:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

12. Indexing the elements

You can write an indexed for loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
  // statement about index i
```

Example: find the maximum element in the vector, and where it occurs.

Code:

```
// array/vectoridxmax.cpp
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i=0; i<numbers.size(); ++i) {
  int v = numbers.at(i);
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
println( "Max: {} at index: {}",
          tmp_max,tmp_idx );
```

Output:

```
Max: 6.6 at index: 3
```

13. Do Not Repeat Yourself

- A `vector` ‘knows’ its size;
- storing the size in a variable is redundant;
- ⇒ do not repeat yourself; use the `size` method.

14. A philosophical point

Conceptually, a `vector` can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

Exercise 2

Normalization:

```
vector<float> x; // initialize!
normalize(x);
cout << norm(x); // should give 1
```

- Write the `norm` and `normalize` functions.

$$\|x\| = \sqrt[2]{\sum x_i^2}$$

- Is there another possibility for `normalize` than the above? Arguments for/against?

Exercise 3

Use general *p*-norm:

$$\|x\|_p = \sqrt[p]{\sum x_i^p}$$

- Read the value of *p*;
- Change the syntax to

```
normalize( x,norm_function )
```

  where `norm_function` computes the *p*-norm. (Hint: use a lambda expression so that you can call

```
norm_function( x );
```

15. Vector copy

Vectors can be copied just like other datatypes:

```
Code:

// array/vectorcopy.cpp
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
println( "{},{}",v.at(2),vcopy.at(2)
      );
```

```
Output:

3.5,7
```

16. Vector methods

A vector is an object, with methods.

Given `vector<sometype> x`:

- Get elements, including bound checking, with `ar.at(3)`.
  Note: zero-based indexing.
  (also get elements with `ar[3]`: see later discussion.)
- Size: `ar.size()`.
- Other functions: `front`, `back`, `empty`, `push_back`.
- With iterators (see later): `insert`, `erase`

17. Your first encounter with templates

`vector` is a ‘templated class’: `vector`<*X*> is a vector-of-*X*.

Code behaves as if there is a class definition for each type:

```
class vector<int> {
public:
  size(); at(); // stuff
}
```

```
class vector<float> {
public:
  size(); at(); // stuff
}
```

Actual mechanism uses templating: the type is a parameter to the class definition.

Dynamic behaviour

18. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

```
// array/vectorend.cpp
vector<int> mydata(5,2);
// last element:
println( "{}",mydata.back() );
mydata.push_back(35);
println( "{}",mydata.size() );
// last element:
println( "{}",mydata.back() );
```

Output:

```
6
35
```

Similar functions: `pop_back`, `insert`, `erase`.

Flexibility comes with a price.

19. When to push back and when not

Known vector size:

```
int n = get_inputsize();
vector<float> data(n);
for ( int i=0; i<n; i++ ) {
  auto x = get_item(i);
  data.at(i) = x;
}
```

Unknown vector size:

```
vector<float> data;
float x;
while ( next_item(x) ) {
  data.push_back(x);
}
```

If you have a guess as to size: `data.reserve(n)`.

(Issue with array-of-object: in left code, constructors are called twice.)

20. Filling in vector elements

You can push elements into a vector:

```
// array/arraytime.cpp
vector<int> flex;
/* ... */
for (int i=0; i<LENGTH; ++i)
  flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat.at(i) = i;
```

21. Filling in vector elements

With subscript:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat[i] = i;
```

You can also use `new` to allocate*:

```
// array/arraytime.cpp
int *stat = new int[LENGTH];
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat[i] = i;
```

*Considered bad practice. Do not use.

22. Timing the ways of filling a vector

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

Vectors and functions

23. Vector as function return

You can have a vector as return type of a function.

Example: this function creates a vector, with the first element set to the size:

Code:

```
// array/vectorreturn.cpp
vector<int> make_vector(int n) {
  vector<int> x(n);
  x.at(0) = n;
  return x;
}
    /* ... */
  vector<int> x1 = make_vector(10);
  // ”auto” also possible!
  cout << "x1 size: "
       << x1.size() << '\n';
  cout << "zero element check: "
       << x1.at(0) << '\n';
```

Output:

```
x1 size: 10
zero element check: 10
```

24. Vector as function argument

You can pass a vector to a function:

```
double slope( vector<double> v ) {
  return v.at(1)/v.at(0);
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

25. Vector pass by value example

Code:

```
// array/vectorpassnot.cpp
void set0
  ( vector<float> v,float x )
{
  v.at(0) = x;
}
    /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

Output:

```
3.5
```

- Vector is copied
- ‘Original’ in the calling environment not affected
- Cost of copying?

26. Vector pass by reference

If you want to alter the vector, you have to pass by reference:

Code:

```
// array/vectorpassref.cpp
void set0
  ( vector<float> &v,float x )
{
  v.at(0) = x;
}
    /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

Output:

```
4.6
```

- Parameter vector becomes alias to vector in calling environment ⇒ argument *can* be affected.
- No copying cost
- What if you want to avoid copying cost, but need not alter the argument?

27. Vector pass by const reference

Passing a vector that does not need to be altered:

```
int f( const vector<int> &ivec ) { ... }
```

- Zero copying cost
- Not alterable, so: safe!
- (No need for pointers!)

Exercise 4

Revisit exercise **??** and introduce a function for computing the L_2 norm.

(hints for the next exercise)

Random numbers:

```
// high up in your code:
#include <random>
using std::rand;

// in your main or function:
float r = 1.*rand()/RAND_MAX;
// gives random between 0 and 1
```

(You will learn a better random later)

Exercise 5

Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
vector<float> sorted = sort(values);
```

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place
(overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)