Arrays and Vectors

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: October 10, 2024

1. 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.

2. Accessing vector elements

Square bracket notation (zero-based):

Code:

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

Output:

```
4,8
```

With bound checking:

Code:

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

Output:

```
4,8
```

3. Vector elements out of bounds

Square bracket notation:

Code:

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

Output:

```
1,4
```

With bound checking:

Code:

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

Output:

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

4. Short vectors

Short vectors can be created by enumerating their elements:

```
1 // array/shortvector.cpp
2 #include <vector>
3 using std::vector;
4
5 int main() {
6   vector<int> evens{0,2,4,6,8};
7   vector<float> halves = {0.5, 1.5, 2.5};
8   auto halfloats = {0.5f, 1.5f, 2.5f};
9   cout << evens.at(0)
10       << " from " << evens.size()
11       << '\n';
12  return 0;
13 }
```

Exercise 1

1. Take the above snippet, compile, run.
2. Add a statement that alters the value of a vector element. Check that it does what you think it does.
3. Add a vector of the same length as the `evens` vector, containing odd numbers which are the even values plus 1?

You can base this off the file `shortvector.cpp` in the repository

5. 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.

6. 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
```

7. Range over elements

Finding the maximum element

Code:

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

Output:

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

Exercise 2

Find the element with maximum absolute value in a vector. Use:

```
vector<int> numbers = {1,-4,2,-6,5};
```

Hint:

```
#include <cmath>
..
absx = abs(x);
```

Exercise 3

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.

8. Range over vector denotation

Code:

```
1 // array/rangedenote.cpp
2 for ( auto i : {2,3,5,7,9} )
3   cout << i << ",";
4 cout << '\n';
```

Output:

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

9. 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:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2)
7      << '\n';
```

Output:

```
6.6
```

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

10. Example: multiplying elements

Example: multiply all elements by two:

Code:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2)
7      << '\n';
```

Output:

```
6.6
```

11. 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:

```
1 // array/vectoridxmax.cpp
2 int tmp_idx = 0;
3 int tmp_max = numbers.at(tmp_idx);
4 for (int i=0; i<numbers.size(); ++i) {
5   int v = numbers.at(i);
6   if (v>tmp_max) {
7     tmp_max = v; tmp_idx = i;
8   }
9 }
10 cout << "Max: " << tmp_max
11      << " at index: " << tmp_idx << '\n';
```

Output:

```
Max: 6.6 at
    ↪index: 3
```

12. On finding the max and such

For many such short loops there are ‘algorithms’; see later.

13. 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 4

Find the location of the first negative element in a vector.

Which mechanism do you use?

Exercise 5

Create a `vector` x of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in L_2 norm and check the correctness of your calculation, that is,

1. Compute the L_2 norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

14. Indexing with pre/post increment

Indexing in `while` loop and such:

```
x = a.at(i++); /* is */ x = a.at(i); i++;
y = b.at(++i); /* is */ i++; y = b.at(i);
```

15. Vector copy

Vectors can be copied just like other datatypes:

Code:

```
1 // array/vectorcopy.cpp
2 vector<float> v(5,0), vcopy;
3 v.at(2) = 3.5;
4 vcopy = v;
5 vcopy.at(2) *= 2;
6 cout << v.at(2) << ","
7      << vcopy.at(2) << '\n';
```

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:

```
1 // array/vectorend.cpp
2 vector<int> mydata(5,2);
3 // last element:
4 cout << mydata.back()
5      << '\n';
6 mydata.push_back(35);
7 cout << mydata.size()
8      << '\n';
9 // last element:
10 cout << mydata.back()
11      << '\n';
```

Output:

```
2
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:

```
1 // array/arraytime.cpp
2 vector<int> flex;
3 /* ... */
4 for (int i=0; i<LENGTH; ++i)
5   flex.push_back(i);
```

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

```
1 // array/arraytime.cpp
2 vector<int> stat(LENGTH);
3 /* ... */
4 for (int i=0; i<LENGTH; ++i)
5   stat.at(i) = i;
```

21. Filling in vector elements

With subscript:

```
1 // array/arraytime.cpp
2 vector<int> stat(LENGTH);
3 /* ... */
4 for (int i=0; i<LENGTH; ++i)
5   stat[i] = i;
```

You can also use `new` to allocate*:

```
1 // array/arraytime.cpp
2 int *stat = new int[LENGTH];
3 /* ... */
4 for (int i=0; i<LENGTH; ++i)
5   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
```

23. Size and capacity

Grow by pushing elements:

Code:

```
1  // array/grow.cpp
2    vector<int> data;
3    const int up=10;
4    for ( int i=0; i<=up; ++i ) {
5      cout << "size=" << data.size()
6           << ", capacity=" <<
       data.capacity()
7           << '\n';
8      if (i==up) break;
9      data.push_back(i);
10   }
```

Output:

```
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=6, capacity=8
size=7, capacity=8
size=8, capacity=8
size=9, capacity=16
size=10, capacity=16
```

24. Size and capacity

Grow by resizing:

Code:

```
1 // array/grow.cpp
2 cout << "Resizing:\n";
3 data.resize( data.size()+7 );
4 cout << "size=" << data.size()
5      << ", capacity=" <<
       data.capacity()
6      << '\n';
```

Output:

```
Resizing:
size=17, capacity=20
```

Vectors and functions

25. 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:

```
1 // array/vectorreturn.cpp
2 vector<int> make_vector(int n) {
3   vector<int> x(n);
4   x.at(0) = n;
5   return x;
6 }
7     /* ... */
8   vector<int> x1 = make_vector(10);
9   // "auto" also possible!
10  cout << "x1 size: "
11       << x1.size() << '\n';
12  cout << "zero element check: "
13       << x1.at(0) << '\n';
```

Output:

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

26. 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.

27. Vector pass by value example

Code:

```
1 // array/vectorpassnot.cpp
2 void set0
3   ( vector<float> v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v.at(0) << '\n';
```

Output:

```
3.5
```

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

28. Vector pass by reference

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

Code:

```
1 // array/vectorpassref.cpp
2 void set0
3   ( vector<float> &v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  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?

29. 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 6

Revisit exercise 5 and introduce a function for computing the L_2 norm.