Arrays and Vectors

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: August 28, 2022

1: Vectors

1. What are vectors?

- Contiguous, indexed, storage of items:
  (often called 'array' but that has other meanings)
  - items of any type (but the same for all elements of one vector)
  - potentially very many items
- Indexed set of items
- . . . but if you don't need the index: collection of items

2. Short vectors

Short vectors can be created by enumerating their elements:

```
#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) << "\n";
  return 0;
}
```

Exercise 1

1. Take the above snippet, supply the missing header lines, 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, containing odd numbers, which are the even values plus 1?

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

3. Range over elements

You can write a range-based for loop, which considers the elements as a collection.

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

Code:

```
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

```
[array] dynamicmax:

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);
```

4. Range over vector denotation

Code:

```
for ( auto i : {2,3,5,7,9} )
  cout << i << ",";
cout << "\n";
```

Output

```
[array] rangedenote:

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

5. 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.

6. Accessing vector elements

Square bracket notation (zero-based):

Code:

```
vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
     << numbers[1] << "\n";
```

Output

```
[array] assignbracket:

4,8
```

With bound checking:

Code:

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

Output

```
[array] assignatfun:

4,8
```

Safer, slower.

(Remember Knuth about optimization.)

7. Vector elements out of bounds

Square bracket notation:

```
Code:

vector<int> numbers = {1,4};
numbers[-1] += 3;
numbers[2] = 8;
cout << numbers[0] << ","
     << numbers[1] << "\n";
```

```
Output
[array]
assignoutofboundbracket:

1,4
```

With bound checking:

```
Code:

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

```
Output
[array]
assignoutofboundatfun:

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

8. 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:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2) << "\n";
```

Output

```
[array] vectorrangeref:

6.6
```

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

9. 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:

```
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;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx <<
     "\n";
```

Output

```
[array] vecidxmax:

Max: 6.6 at index: 3
```

10. 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 3

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

Which mechanism do you use?

Exercise 4

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?

11. 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);
```

12. Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << "\n";
```

Output

```
[array] vectorcopy:

3.5,7
```

13. 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`.
- With iterators (see later): `insert`, `erase`

14. 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. More later.

2: Dynamic behaviour

15. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << "\n";
cout << mydata.back();
     << "\n";
```

Output

```
[array] vectorend:

6
35
```

Similar functions: `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

16. 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)`.

17. Filling in vector elements

You can push elements into a vector:

```
  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:

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

18. Filling in vector elements

With subscript:

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

You can also use `new` to allocate*:

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

*Considered bad practice. Do not use.

19. 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
```

3: Vectors and functions

20. 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.

21. Vector pass by value example

Code:

```
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

```
[array] vectorpassnot:

3.5
```

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

22. Vector pass by reference

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

Code:

```
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

```
[array] vectorpassref:

4.6
```

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

23. 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 5

Revisit exercise 4 and introduce a function for computing the L_2 norm.

24. 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:

```
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

```
[array] vectorreturn:

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

Exercise 6

Write code to take a vector of integers, and construct two vectors, one containing all the odd inputs, and one containing all the even inputs. So:

input:

```
5,6,2,4,5
```

output:

```
5,5
6,2,4
```

Can you write a function that accepts a vector and produces two vectors as described?

(hints for the next exercise)

```
// 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
```

Exercise 7

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.)

4: Vectors in classes

25. Can you make a class around a vector?

You may want a class of objects that contain a vector. For instance, you may want to name your vectors.

```
class named_field {
private:
  vector<double> values;
  string name;
```

The problem here is when and how that vector is going to be created.

26. Create the contained vector

Use initializers for creating the contained vector:

```
class named_field {
private:
  string name;
  vector<double> values;
public:
  named_field( string name,int nelements )
    : name(name),
      values(vector<double>(n)) {
  };
};
```

Less desirable method is creating in the constructor:

```
  named_field( string uname,int nelements ) {
    name = uname;
    values = vector<double>(n);
  };
```

5: Multi-dimensional arrays

27. Multi-dimensional vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

Create a row vector, then store 10 copies of that:
vector of vectors.

28. Matrix class

```
class matrix {
private:
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
```

Exercise 8

Write `rows()` and `cols()` methods for this class that return the number of rows and columns respectively.

Exercise 9

Write a method `void set(double)` that sets all matrix elements to the same value.

Write a method `double totalsum()` that returns the sum of all elements.

Code:

```
A.set(3.);
cout << "Sum of elements: "
     << A.totalsum() << "\n";
```

Output

```
[array] matrixsum:

Sum of elements: 30
```

You can base this off the file `matrix.cxx` in the repository

29. Matrix class; better design

Better idea:

```
class Matrix {
private:
  int rows,cols;
  vector<double> elements;
private:
  Matrix( int m,int n )
  : rows(m),cols(n),
    elements(vector<double>(rows*cols))
    {};
  ...
  double get(int i,int j) {
    return elements.at(i*cols+j);
  }
```

(Old-style solution: use cpp macro)

Exercise 10

In the matrix class of the previous slide, why are `m,n` stored explicitly, and not in the previous case?

Exercise 11

Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

30. Pascal's triangle

Pascal's triangle contains binomial coefficients:

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

The coefficients can be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} & \end{cases}$$

(There are other formulas. Why are they less preferable?)

Exercise 12

- Write a class `pascal` so that `pascal(n)` is the object containing n rows of the above coefficients. Write a method `get(i,j)` that returns the (i, j) coefficient.
- Write a method `print` that prints the above display.
- First print out the whole pascal triangle; then:
- Write a method `print(int m)` that prints a star if the coefficient modulo m is nonzero, and a space otherwise.

```
            *
           * *
          *   *
         * * * *
        *       *
       * *     * *
      *   *   *   *
     * * * * * * * *
    *               *
   * *             * *
```

The object needs to have an array internally. The easiest solution is

Optional exercise 13

Extend the Pascal exercise:
Optimize your code to use precisely enough space for the coefficients.

Turn it in!

- Write a program that accepts:
  1. one integer: the height of the triangle. You should use this to construct a `PascalTriangle` object that contains the binomial coefficients. Then:
  2. a number of modulos with which to print the triangle. A value of zero indicates that your program should stop.

  The tester will search for stars in your output and test that you have the right number in each line.
- If you have compiled your program, do a test run:

  ```
  coe_pascal yourprogram.cc
  ```
- Is it Submit if it is correct:

  ```
  coe_pascal -s yourprogram.cc
  ```
- If you don't manage to get your code working correctly, you can submit as incomplete with

  ```
  coe_pascal -i yourprogram.cc
  ```

6: Static arrays

Array creation

C-style arrays still exist,

```
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << "\n";
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << "\n";
}
```

but you shouldn't use them.
Prefer to use `array` class (not in this course)
or `span` (C++20; very advanced)

Ranging

You can range over static arrays same as for `vector`

7: Advanced material

Iterators

31. Beyond begin/end

- An iterator is a little like a pointer
  (into anything iteratable)
- *beginend*
- pointer-arithmetic and 'dereferencing':

```
auto element_ptr = my_vector.begin();
element_ptr++;
cout << *element_ptr;
```

- allows operations (erase, insert) on containers

32. Erase at/between iterators

Erase from start to before-end:

Code:

```
vector<int> counts{1,2,3,4,5,6};
vector<int>::iterator second =
    counts.begin()+1;
auto fourth = second+2; // easier
    than 'iterator'
counts.erase(second,fourth);
cout << counts[0] << "," << counts[1]
    << "\n";
```

Output

```
[iter] erase2:

1,4
```

(Also single element without end iterator.)

33. Insert at iterator

Insert at iterator: value, single iterator, or range:

Code:

```
vector<int>
     counts{1,2,3,4,5,6},zeros{0,0};
auto after_one = zeros.begin()+1;
zeros.insert(
     after_one,counts.begin()+1,counts.begin()+3
     );
//vector<int>::insert(
     after_one,counts.begin()+1,counts.begin()+3
     );
cout << zeros[0] << "," << zeros[1]
     << ","
     << zeros[2] << "," << zeros[3]
     << "\n";
```

Output

```
[iter] insert2:

0,2,3,0
```