Arrays and Vectors

Victor Eijkhout, Harika Gurram,
Je'aime Powell, Charley Dey

Fall 2018

Vectors

Vector definition

Definition, mostly without initialization.

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

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

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.

Accessing vector elements

You have already seen the square bracket notation:

```
vector<double> x(5, 0.1 );
x[1] = 3.14;
cout << x[2];
```

Alteratively:

```
x.at(1) = 3.14;
cout << x.at(2);
```

Safer, slower.

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 and where it occurs.

Code:

```
int tmp_idx = 0;
int tmp_max = numbers[tmp_idx];
for (int i=0; i<5; i++) {
  int v = numbers[i];
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << endl;
```

Output
[array] idxmax:

```
Max: 6 at index: 3
```

Indexing with pre/post increment

Array indexing in `while` loop and such:

```
x = a[i++]; /* is */ x = a[i]; i++;
y = b[++i]; /* is */ i++; y = b[i];
```

Range over elements

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

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

Code:

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output
[array] dynamicmax:

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

Ranging over a vector

```
for ( auto e : my_vector)
  cout << e;
```

Note that e is a copy of the vector element:

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output
[array]
vectorrangecopy:

```
3.3
```

Ranging over a vector by reference

To set array elements, make `e` 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[2] << endl;
```

Output
[array] vectorrangeref:

```
6.6
```

Vector initialization

You can initialize a vector as a whole:

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

(This syntax requires compilation with the `-std=c++11` option.)

Vector initialization'

There is a syntax for initializing a vector with a constant:

```
vector<float> x(25,3.15);
```

which gives a vector of size 25, with all elements initialized to 3.15.

Vector indexing

Your choice: fast but unsafe, or slower but safe

```
vector<double> x(5);
x[5] = 1.; // will probably work
x.at(5) = 1.; // runtime error!
```

Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v[2] = 3.5;
vcopy = v;
cout << vcopy[2] << endl;
```

Output
[array] vectorcopy:

```
./vectorcopy
3.5
```

Exercise 1

Find the element with maximum absolute value in an array. Use:

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

Which mechanism do you use for traversing the array?

Hint:

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

Exercise 2

Find the location of the first negative element in an array.

Which mechanism do you use?

Exercise 3

Create a `vector` x of `float` elements, and set them to random values.

Now normalize the vector in L_2 norm, that is, scale each element by the same coefficient α so that $\sum_i x_i^2 = 1$. Check the correctness of your calculation.

Vector methods

- Get elements with `ar[3]` (zero-based indexing).
  (for C programmers: this is not dereferencing, this uses an operator method)
- Get elements, including bound checking, with `ar.at(3)`.
- Size: `ar.size()`.
- Other functions: `front`, `back`.
- `vector` is a 'templated class'

Dynamic extension

Extend with `push_back`:

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output
[array] vectorend:

```
6
35
```

also `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

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

Vector of vectors.

Static arrays

Array creation

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

Ranging

Same as for `vector`

Dynamic behaviour

Dynamic size extending

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

Vector extension

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;
```

Vector extension

With subscript:

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

You can also use `new` to allocate (see section **??**):

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

Timing

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

Exercise 4

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
```

Vectors and functions

Vector as function return

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

Code:

```
vector<int> make_vector(int n) {
  vector<int> x(n);
  x[0] = n;
  return x;
}
  /* ... */
  vector<int> x1 = make_vector(10); // "auto" also possible!
  cout << "x1 size: " << x1.size() << endl;
  cout << "zero element check: " << x1[0] << endl;
```

Output
[array] vectorreturn:

```
./vectorreturn
x1 size: 10
zero element check: 10
```

Vector as function argument

You can pass a vector to a function:

```
void print0( vector<double> v ) {
  cout << v[0] << endl;
};
```

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

Vector pass by value example

Code:

```
void set0
  ( vector<float> v,float x )
{
  v[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output
[array] vectorpassnot:

```
./vectorpassnot
3.5
```

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[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output
[array] vectorpassref:

```
./vectorpassref
4.6
```

Exercise 5

Revisit exercise 16 and introduce a function for computing the L_2 norm.

Exercise 6

Revisit exercise 28.

Can you write a function that accepts a vector and returns two vectors with the above functionality?

(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);
sort(values);
```

(This creates a vector of random values of a specified length, and then sorts it.)

See section **??** for the random fuction.

Vectors in classes

Can you make a class around a vector?

Vector needs to be created with the object, so you can not have the size in the class definition

```
class witharray {
private:
  vector<int> the_array( ???? );
public:
  witharray( int n ) {
    thearray( ???? n ???? );
  }
}
```

Create and assign

The following mechanism works:

```
class witharray {
private:
  vector<int> the_array;
public:
  witharray( int n )
    : the_array(vector<int>(n)) {
  };
};
```

Better than

```
  witharray( int n ) {
    the_array = vector<int>(n);
  };
```

Matrix class

```
class matrix {
private:
  int rows,cols;
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    rows = m; cols = 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);
  };
};
```

Matrix class'

Better idea:

```
elements = vector<double>(rows*cols);
...
void get(int i,int j) {
  return elements.at(i*cols+j);
}
```

(Even more efficient: use cpp macro)

Exercise 8

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

Implement matrix-matrix multiplication.

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 easily be computed from the recurrence

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

Exercise 9

- 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.
- 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 to make an array of size $n \times n$.

Bonus: when you have that code working, optimize your code to